MIT/LCS/TR-174

A CASE STUDY OF INTERMODULE DEPENDENCIES IN A VIRTUAL MEMORY SUBSYSTEM

Douglas H. Hunt

December 1976

This research was supported in part by Honeywell Information Systems Inc., and in part by the United States Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense of the United States under ARPA Order No. 2641, which was monitored by ISTAO under Contract No. F 19628-74-C-0193.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(Formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

ACKNOWLEDGEMENTS

First of all, I would like to thank my thesis supervisor, Jerry Saltzer, for providing critical commentary on the drafts of this thesis. During the past few months, I could give him a draft copy on a given day and expect to see his written comments the following morning.

Both Dave Clark and Dave Reed commented on earlier drafts, and for their efforts I am grateful. Dave Reed also cheerfully consented, on many occasions, to help evaluate my evolving thesis ideas.

A number of my colleagues, namely Art Benjamin, Jeff Goldberg, Rick Gumpertz, Phil Janson, Raj Kanodia, and Allen Luniewski offered useful suggestions during the course of the thesis research. More important than that, they provided interesting lunchtime discussions during our Subcommittee meetings, and comradery on the volleyball court.

Finally, I would like to thank Tom Fuller, who, aided by the wisdom of H. W. Fowler and Wilson Follett, carefully reviewed my use and abuse of the English language in this thesis.

This research was supported in part by Honeywell Information Systems Inc., and in part by the United States Air Force Information Systems Technology Applications Office (ISTAO) and the Advanced Research Projects Agency (ARPA) of the Department of Defense of the United States under ARPA Order No. 2641, which was monitored by ISTAO under Contract No. F 19628-74-C-0193.

A CASE STUDY OF INTERMODULE DEPENDENCIES IN A VIRTUAL MEMORY SUBSYSTEM*

by

Douglas Hamilton Hunt

ABSTRACT

A problem currently confronting computer scientists is to develop a method for the production of large software systems that are easy to understand and certify. The most promising methods involve decomposing a system into small modules in such a way that there are few intermodule dependencies. In contrast to previous research, this thesis focuses on the nature of the intermodule dependencies, with the goal of identifying and eliminating those that are found to be unnecessary. Using a virtual memory subsystem as a case study, the thesis describes a structure in which apparent dependencies can be eliminated. Owing to the nature of virtual memory subsystems, many higher level functions can be performed by lower level modules that exhibit minimal interaction. The structuring methods used in this thesis, inspired by the structure of the LISP world of atomic objects, depend on the observation that a subsystem can maintain a copy of the name of an object without being dependent upon the object manager. Since the case study virtual memory subsystem is similar to that of the Multics system, the results reported here should aid in the design of similar sophisticated virtual memory subsystems in the future.

THESIS SUPERVISOR: Jerome H. Saltzer

TITLE: Professor of Computer Science and Engineering

---

*This report is based upon a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, on December 21, 1976 in partial fulfillment of the requirements for the degrees of Master of Science and Electrical Engineer.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

Chapter I

Introduction

1.1 Introduction

This thesis focuses on the interactions between certain components of an operating system. Our goal is to show that a nontrivial subsystem, a virtual memory (VM) subsystem, can be organized as a set of modules that are related to each other in particularly simple ways. The thesis presents the view that, owing to the nature of VM subsystems, the supporting modules need interact only in a few stylized ways.

1.2 The Problem

The research reported here is devoted to one aspect of the problem of providing correct and reliable components for large-scale computing systems. The general problem is that it is difficult to maintain and modify, and it is particularly difficult to verify, the correct operation of large general-purpose systems. This general problem is due, in part, to an excessively high degree of connectivity between system modules. The nature of the interaction between two modules is such that the correct operation of one module depends upon the correct operation of the other. These interactions are thus evidence of intermodule dependencies.

This thesis treats one aspect of the general problem by examining intermodule dependencies in the context of a case study virtual memory subsystem. The reason for limiting the problem in this way is to rely upon characteristics that are inherent in VM subsystems to provide guidelines for determining classes of dependencies that may be essential in that context.

The specification of the case study VM is based on the specifications of the Multics [Bensoussan72, Organick72, Multics74], CAL [Sturgis74, Lampson76], Hydra [Wulf74, Jones75], and Stanford Research Institute (SRI) [Neumann75, Robinson75] VM subsystems. These virtual memory subsystems, as well as those of the Plessey 250 [England72], TSS/360 [Lett68], and HITAC 8800 [Nakazawa72] systems, support shared segments or segment-like objects. Each of these subsystems is a relatively complex operating system component. This study of intermodule dependencies in a sophisticated VM subsystem should aid in the design of similar subsystems in the future.

The modular structure of a subsystem should have the following properties: 1) no one module should be particularly large, 2) the interconnections between modules should be simple and few in number, and 3) the intermodule dependencies should form a partial order. The primary goal of this thesis is to show, in the context of the case study, how such a constrained modular structure can be obtained. A secondary goal of the thesis is to introduce some novel methods for obtaining this modular structure that can be applied to other areas of system design.

1.3 Method of Solution

The method of solution relies on established structuring methodologies, as well new methodologies developed during the course of this research. The established methodologies are described in this section; the new ones are described in the next section, which summarizes the results of this thesis.

One approach to achieving a modular structure that can exhibit the three properties above is called the object-oriented approach, in which each module is a subsystem that supports all computational objects of a particular type.

The only way to carry out an operation on a particular type of object is to invoke the corresponding managing subsystem. In general, a module that supports a certain type of object depends on other modules that support other types. Dependencies among these modules are straightforward, since they must correspond to the interfaces of the object manager subsystems.

A second approach for structuring subsystems is the layering approach, in which a subsystem is regarded as an ordered set of layers, or abstract machines, such that each layer uses the environment provided by layers below it in the ordering to implement a more sophisticated abstract machine. The important characteristic of this approach is that since each layer is designed to operate in an environment provided only by those layers below it, no layer embodies knowledge of any higher layer. Assuming that layers are separated by protection barriers, the correct operation of a given layer depends only on that of lower layers.

These two approaches illustrate different aspects of modular structure. A system organized according to the object-oriented approach appears to be a collection of data abstractions. Modular structure is achieved because, by assumption, the procedures that are most likely to interact strongly are precisely those that serve to produce the same data abstraction. These procedures are grouped within distinct type managers. Hence the object-oriented approach should yield a structure in which intermodule invocations are relatively infrequent. A layered system appears to be a collection of progressively more sophisticated abstract machines. In this case, modular structure is achieved because, by assumption, the difference between any two adjacent machines in the ordering is rather small. Therefore a module comprising only a small collection of procedures should be sufficient to

produce a given abstract machine from the preceding one. The layering approach should yield a structure consisting of small modules.

The object-oriented and layering design methodologies are not incompatible; a subsystem can have a structure that is both object-oriented and layered. The most straightforward intersection of these two methodologies results in regarding each module as both a manager of an object type and a layer. In this thesis we do not insist on a one-to-one correspondence; for example, some case study VM layers contain several object manager subsystems.

It is possible to structure a subsystem by either of these methodologies so that intermodule dependencies form a partial order. If the dependencies form a partial order, then the process of verifying correct operation can be decomposed in a natural way. Since there are no dependency loops, there must exist modules that depend on no others. The correct operation of these modules is verified first. Thereafter, any module that depends only on already-verified modules can be verified. (1) To retain this structure that is desirable for verification, we shall insist on a dependency relationship among VM modules that is a partial ordering. Since the term "layer" connotes a total ordering, we prefer to use the term region. However, wherever the context is not sufficiently restrictive, we will use these terms interchangeably.

The intermodule dependencies are classified according to their effects. If a specification of module A that does not take time into account can be violated by incorrect operation of module B, we say that A has a strong

---

(1) Dependency loops among modules complicate the verification process. A technique for breaking dependency loops, called sandwiching, is described by Parnas [Parnas76].

dependency on B. If A does not have a strong dependency on B, but a specification of A that takes time into account can be violated by incorrect operation of B, we say that A has a weak dependency on B. For example, in a layered subsystem a lower layer is, by design, not strongly dependent on a higher layer. As another example, two regions that interact only through a shared semaphore are mutually weakly dependent. In this thesis, we are concerned much more with strong dependencies than with weak dependencies, since failure in the case of weakly dependent modules implies only a denial of service. We shall use the term "dependency" by itself to mean "strong dependency", and "independent" to mean "not strongly dependent".

Although some novel notions for implementing a VM subsystem do appear in this thesis, such notions are not an end in themselves. They serve as a means of illustrating a structuring methodology. In fact, there is an assumption in this thesis that the machine architecture supporting the case study VM is rather conventional. As a consequence some tempting but irrelevant mechanisms, often mentioned in footnotes, were not included in this analysis.

The approach taken in this thesis is to understand and classify the dependencies among the case study VM modules. A measure of the success of this approach is the extent to which each dependency can be explicitly justified.

1.4 Results

We observe in this thesis that two kinds of operations are fundamental to the functioning of the case study VM subsystem. The first kind of operation is one that can associate and dissociate two computational objects. The second kind of operation is one that fetches attributes of a computational

object when given its name. We derive two structuring techniques -- one for each kind of operation -- that implement each operation in such a way as to reduce the number of strong intermodule dependencies. These techniques are patterned after the view taken by the designers of LISP [McCarthy62] towards the LISP world of atomic objects. By adapting their point of view to the problem at hand, it becomes simpler to identify superfluous intermodule dependencies.

Corresponding to the operations that associate and dissociate objects are the LISP operations of binding and unbinding. The chief virtue of the binding notion, from the point of view of this thesis, is that it is a one-way relationship: the behavior of object B is unaffected if object A is bound to it. Treating the association between two objects as a binding makes explicit the nature of the dependency between the corresponding subsystems that implement them.

Corresponding to the operation that maps object names to attributes is the LISP notion of a property list. The significant observation concerning a property list is that the module that associates the data therein with an object need place no interpretation on the data. Accordingly, if module C associates an object of module A with an object of module B, its behavior need not depend (except possibly weakly) upon the behavior of module A or module B.

Additionally, we suggest a structuring method that serves the engineering goal of achieving economy of mechanism. We refer to this method as the principle of the greatest common mechanism. Even in subsystems that are well-structured according to the above criteria, a number of modules may contain similar or identical functional subsets. For example, the specification of several case study VM modules reveals that, given an object

name, they return a particular attribute. A supporting module that implements this mapping function for the other VM modules could be provided. If the supporting module were too small, in the sense that its mapping mechanism were not sufficiently general to provide a service for each of the other modules, then economy of mechanism would be sacrificed since at least one of the other modules must implement its own mapping function. If the supporting module were too large, in the sense that it not only provided the necessary mapping functions for the other modules but also provided specialized functions for some of the modules, then the correct operation of all the modules would depend upon the correct operation of these specialized functions. In order that the supporting module not err in either direction, it should provide the greatest common mechanism, which would consist of the intersection of the functionalities required by the dependent modules.

The principle of greatest common mechanism should be distinguished from the principle of _least common mechanism_, originally suggested by M. D. Schroeder and described by Popek [Popek74]. The principle of least common mechanism is relevant to the design of a system that contains a _security kernel_ [Schroeder75]. A security kernel is an encapsulated set of programs and data that implements the security policy of an operating system. Typically, a security policy specifies conditions that must be met before information can pass between two users, or between a data repository and a user. The kernel should allow information flow only when the specified conditions are met. An error in any encapsulated program may allow information flow that is contrary to the policy. One method for reducing the likelihood of such errors is to remove all mechanisms not relevant to security from the kernel. Although such mechanisms may be common mechanisms, they do

not require the high degree of privilege that kernel mechanisms do, and errors in their operation may violate the security policy. The least common mechanism principle states that such mechanisms should be excluded from the security kernel. Popek mentions the subsystem that supports a user I/O interface as an example of a mechanism that is often included in the most privileged part of an operating system, even though it does not require such privilege. According to the principle of least common mechanism, this subsystem should not be included in a security kernel. The principle of least common mechanism is thus a means for reducing the likelihood of unauthorized information flow, whereas the principle of greatest common mechanism is a means for achieving economy of mechanism. These principles are not incompatible. For example, the first principle states that the user I/O facility should be outside the kernel, and the second principle states that it should be a common mechanism. Both objectives can be met if the user I/O facility is a common mechanism, outside the security kernel.

1.5 Related Research

In the past few years both layered and object-oriented general-purpose systems have been built, providing evidence that these structuring techniques can be used in a practical context. More recently researchers have sought to specify and verify the correct operation of system components. As part of these efforts, some researchers have focused on the nature of modules and their interdependencies.

The layered approach to structuring was employed in the development of the "THE" system [Dijkstra68]. More recent examples of layered systems

include the CAL [Lampson76] and Venus [Liskov72] systems, as well as the family of systems described by Parnas [Parnas76].

The object-oriented approach was employed in the CAL system, and more recently in the Hydra system [Wulf74]. A specification for a layered, object-oriented system [Robinson75] has been produced by Robinson and others at SRI. However, none of the object-oriented systems has carried this structuring approach all the way down to the hardware interface; the CAL effort was perhaps the most successful attempt to do so.

In order to verify correct operation, it is necessary first to define it. Naur [Naur66], Floyd [Floyd67] and Hoare [Hoare69] showed that small programs can be proved to satisfy a set of assertions. Researchers at SRI, building upon these efforts and the work of Parnas [Parnas72] in the area of program specification, are developing a methodology for proving properties of larger collections of programs. Among the many kinds of assertions that programs may be shown to satisfy, assertions about the secure operation of systems [Bell74, Neumann75] have probably received most attention.

Assertions made in the following chapters about intermodule dependencies can be justified without recourse to a formal notion of correctness. There is, however, an assumption about correctness that is fundamental to this thesis: the correct operation of a layer may be determined without regard to its use [Habermann76]. For example, an errant program may appear to "misuse" a computer hardware layer if it attempts to divide by zero. However, as long as the hardware behaves as specified in this and in other "erroneous" situations, it is said to be correct.

Currently, there are a number of research efforts under way that are related to the design of verifiable, general-purpose systems. At SRI, a

system design methodology that supports semi-automated correctness proofs is being developed [Robinson75]. In an effort described by Schroeder [Schroeder75], parts of the Multics operating system are being restructured, with the goal of making manual verification possible. Parnas [Parnas76] describes a notion of intermodule dependency that serves as the basis for structuring a family of operating systems.

The SRI system design effort is part of a larger effort to develop a methodology for designing verifiable systems. The operating system design serves as a case study for the methodology. Currently, there are no plans for producing an implementation of the operating system design. A significant feature of the SRI methodology is that proof of correct operation can be carried out as part of the design process. Global assertions, based on the specifications of a high-level module, can be proved before the supporting low-level modules have been specified. Thus certain classes of inconsistencies can be detected as the system is being specified. As part of this effort, the SRI research team has developed semi-automated tools for checking the consistency of module specifications.

The Computer Systems Research Division of the M.I.T. Laboratory for Computer Science is nearing completion of a project that supports development of a certifiable security kernel for the Multics system. (1) One activity in the scope of this project involves a restructuring of the system software that manages processor and memory resources. As part of this effort Reed [Reed76] has described a design to simplify the management of processor resources in

---

(1) Other participants in this project, sponsored by the Air Force Electronic Systems Division, include Honeywell Information Systems Inc., the MITRE Corporation, and Stanford Research Institute.

Multics. In this design, the processor management function is distributed over two layers. The virtual processor abstraction provided by the lower layer can be used to structure the system supervisor. Janson [Janson76] describes a way of restructuring the Multics virtual memory so that the resulting modules are 1) responsible for managing distinct data abstractions, and 2) small enough to be subject to manual verification. Huber [Huber76] has shown how the use of dedicated virtual processors can simplify the structure of the Multics virtual memory. These restructuring efforts are all aimed at establishing a more coherent layered structure within the Multics supervisor. A distinctive aspect of this project is the emphasis on viability: the restructuring efforts must be carried out within the constraints imposed by a commercially available system.

Work on a family of operating systems by Parnas and his colleagues represents the first total system design effort in which intermodule dependencies have received careful scrutiny. Modules interact according to the "uses" relationship: module A uses module B if A, in order to function correctly, requires the presence of a correct version of B. The system structure can be represented as a directed acyclic graph whose edges correspond to the "uses" relationship. The "uses" relationship is thus a partial ordering among modules. Assuming that assertions about correct operation in the Parnas family of systems include assertions about elapsed time, then the "uses" relation contains the strong dependency relation, but is contained by the union of the strong and weak dependency relations. For example, there are weak dependency relations that are not "uses" relations: if, by invoking module B, module A can cause another entry to be put into a hash table that is maintained by B, then A may be able to cause long hash

table searches and resulting degraded service from module B. In this case B is weakly dependent on A but does not "use" A in the sense of Parnas.

In contrast to previous work, this thesis investigates conditions under which a module simply maintains bindings to objects, without any embedded knowledge of the semantics of the objects. If these conditions are satisfied, then the module maintaining the bindings cannot be strongly dependent upon those modules that implement the objects.

1.6 Plan of the Thesis

The specifications of the case study virtual memory subsystem are presented in chapter II. First, the notions of an abstract object and an abstract type manager are reviewed. The next section of the chapter describes the capability and access control list models of protection, with an emphasis on the latter model. Additional terms are then defined, and assumptions that further refine the scope of the problem are stated. The final sections of the chapter deal with particular issues relating to type extension in the case study VM, including the relationship of access control lists to extended type objects, and the treatment of directories as extended type objects.

Chapter III is devoted to the lower layers of the case study VM. The mechanisms in these lower layers multiplex main memory. We present an abstract model of a memory multiplexing implementation, based on the manipulation of bindings between autonomous, low-level objects. Each type of low-level object is shown to be related to the other types in a simple way that follows directly from the description of the memory multiplexing model. Although one application of the simple model provides a correspondingly simple VM environment, additional (i.e. recursive) applications provide progressively

more sophisticated environments that approximate segmented address spaces. The dependencies among the VM modules that support this recursive design are analyzed and justified.

In chapter IV, the higher layers of the VM are considered. These layers are intended to support objects with implementation-independent names and with an arbitrary collection of attributes, as well as to support dynamic type extension. To provide confidence in the viability of the VM structure, there is a section of chapter IV devoted to a description of how this VM structure could support any of several possible segmented addressing environments. An important common function of the higher VM layers is the mapping of object names to object attributes. A layer that serves as a common mechanism for supporting these mappings is described. Applying the LISP-inspired property list notion and the principle of greatest common mechanism to the specifications of the higher VM layers results in a design that eliminates unnecessary dependencies.

Chapter V presents a summary of the research reported in this thesis. The assumptions underlying the structuring methods of the thesis are reviewed, together with the particular characteristics of a VM subsystem that have made them readily applicable. We contrast the structuring approaches of chapters III and IV, showing why the former is more appropriate for the lower VM layers and the latter is more appropriate for the higher layers. Following this summary of the thesis results, we mention some problems that remain, and offer suggestions for further research.

Chapter II

The Case Study Virtual Memory Subsystem

2.1 Introduction

It is the purpose of this chapter to specify the case study virtual memory (VM) subsystem. This specification will serve as the context for the analysis given in later chapters. The case study VM has not been implemented; rather it is a design inspired by existing VM subsystems, namely those of Multics [Bensoussan72], CAL [Lampson76], and Hydra [Wulf74], as well as the VM of the system being designed at SRI [Robinson75].

Since the VM is expected to support extended type objects, we begin by reviewing the notion of type extension. We then describe how extended type objects can be protected by access control lists. Throughout this chapter we refer to related work on extensible systems and protection. We include a brief description of the representation of authority hierarchies, and of the treatment of directories as extended type objects. The next two chapters introduce methods for structuring an implementation of this case study VM subsystem.

2.2 Types and Type Extension

This section reviews the notions of object types and abstract type managers. These notions are fundamental to the object-oriented structuring methodology employed in this thesis.

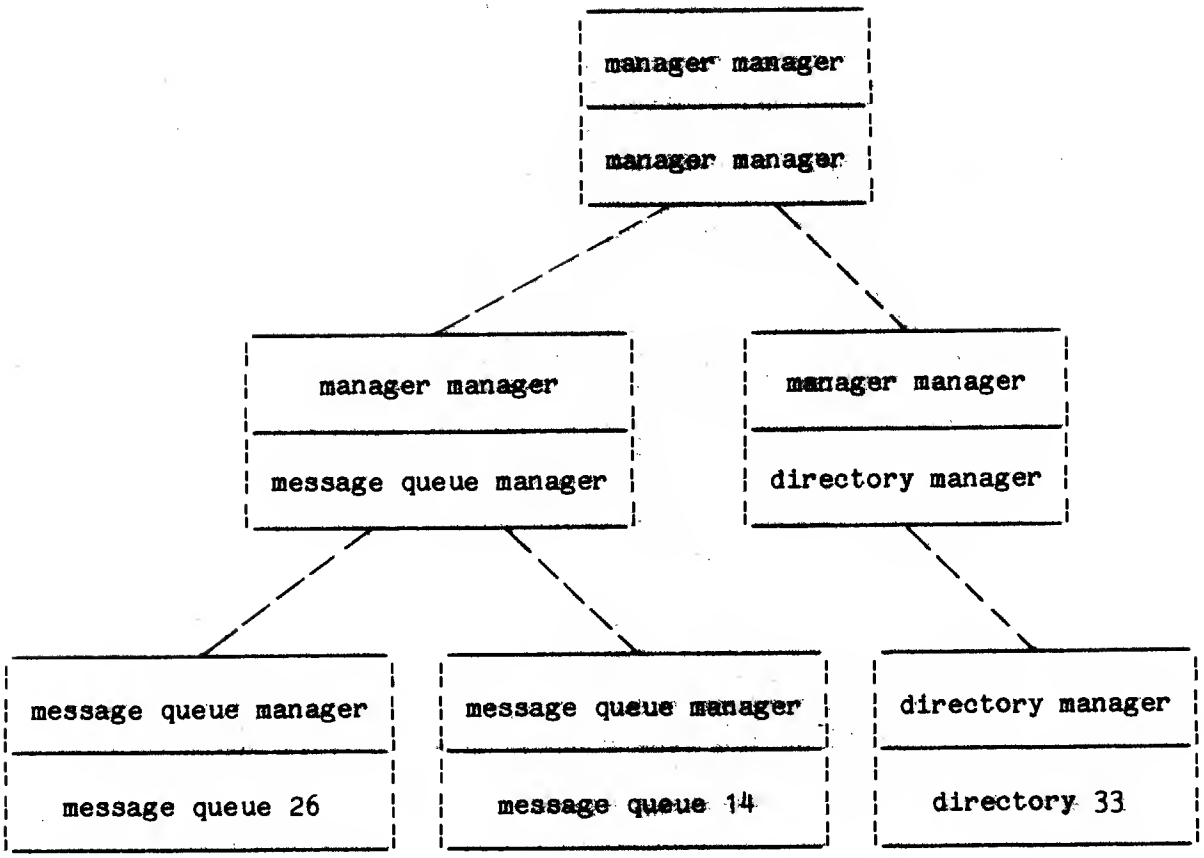
The entities that are manipulated during the course of a computation are called objects. Each object is defined by a set of operations. For example, a "message queue" object might be defined by the operations "enqueue" and

"dequeue". The set of objects can be partitioned into equivalence classes, such that objects with the same set of operations form a class. Each equivalence class is called a type. Thus "message queue" is the name of a type, and there may be many message queue objects that are instances of this type.

Associated with each object type is a subsystem called an abstract type manager, or type manager. A type manager comprises a set of procedures that implement each of the operations of a type. For example, an "enqueue" request for a given message queue object would be directed to the message queue type manager. The type manager would then manipulate the data that represents the given message queue object in such a way that the enqueue operation is effected.

There are two particular operations, "create" and "delete", that are applicable to many object types. Since it is difficult to regard these operations as ones that affect the state of an object, we consider them instead to be operations on the type manager itself. This suggests that the type manager is an object in its own right.

The view that "types are objects" was introduced by Jones [Jones73] and is summarized here. According to this view, the set of all objects forms a tree that is three levels deep. Figure 2-1 is an illustration of the three-level tree of objects. A given object has, among other attributes, a name and a type. The type attribute is generally the name of a different object, one level higher in the tree, that is the type manager for the given object. Leaf nodes, such as "message queue 26", correspond to instances of a type. Interior nodes of the tree, such as "message queue manager", correspond to type managers. The root node corresponds to a special object that has a



Each node in the tree represents one object.
The upper half of each node contains the type of the object,
and the lower half contains the name.

Figure 2-1

The Three-Level Tree of Object Types

type attribute equal to its own name attribute. The root node object is, in effect, a manager of type managers. It is not only possible to create and delete instances of objects, but it is also possible to create and delete type managers. Creation of a new object instance is accomplished by invoking the "create" operation of a type manager. Similarly, creation of a new type manager is accomplished by invoking the "create" operation of the root node object. A system that provides for creation of type managers (i.e. the definition of new types) is said to support dynamic type creation. The CAL, Hydra, and SRI systems all support dynamic type creation. The VM of this thesis also supports dynamic type creation, not only for the flexibility that this feature provides to users, but more importantly because the VM itself is organized as a set of type managers so much common mechanism for supporting user-defined types already exists.

In general, the representation of an object comprises a set of other objects, and a catalog naming each member of that set. We refer to the objects in the representation as representation objects, component objects, or components, and to the catalog as the map. An object that is built out of other objects is called an extended type object (ETO), and its type manager is called an extended type manager (ETM). Thus the set of extended type managers are a proper subset of the set of abstract type managers. The notion of type extension can be applied recursively, so a component object may itself be an ETO. These relationships are shown in Figure 2-2. Figures 2-1 and 2-2 both depict a tree-structured relationship among objects. These two tree structures are independent. The basis of the relationship in Figure 2-1 is

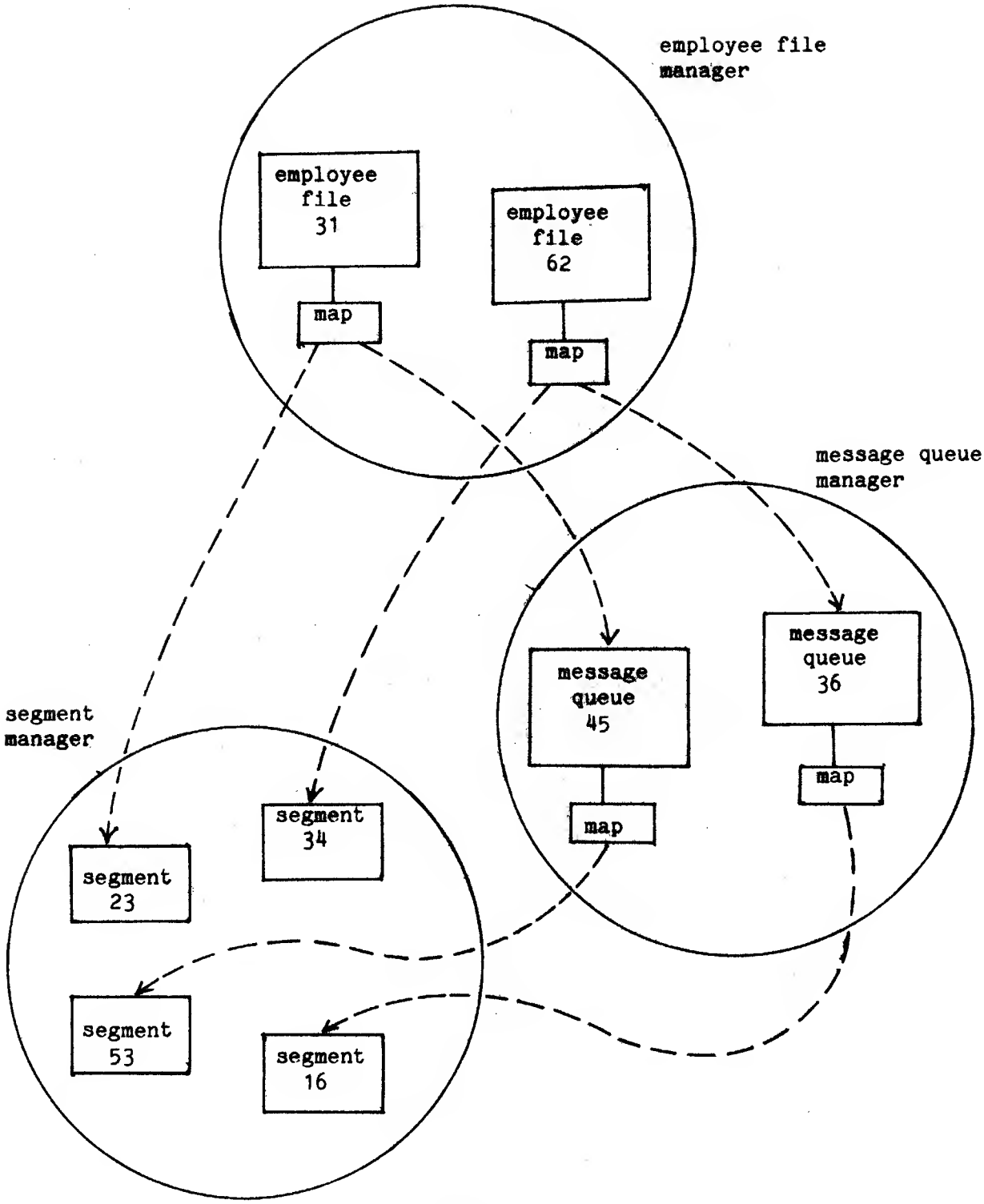


Figure 2-2

Extended Type Objects and Representation Objects

type: each child node corresponds to an instance of the type managed by the parent node. The basis of the relationship in Figure 2-2 is representation: each child node corresponds to a representation object of the parent node.

In order to prevent endless recursion, it is appropriate to regard certain "low level" objects as being atomic. Such objects are called base level objects. Of course, such a distinction is an arbitrary one, since any object in a representation tree could be chosen to a base level object. In the case study VM, it is reasonable to consider segment objects to be base level objects. Even though segments are implemented by more primitive objects, subsystems outside the VM cannot access these primitive components. The segment object was described first by Dennis [Dennis65], and since that time many operating systems have provided segments or similar abstractions. A segment is an ordered collection of storage cells (typically words or bytes) with an associated name. The address of a cell within a segment is an integer that denotes the position of the cell relative to the beginning of the segment. The length of a segment is equal to the number of cells it contains, which may vary during the course of a computation. The name of a segment is location-independent; segment names in the case study VM are non-reusable unique identifiers. Since segments may contain either data or executable instructions, the operations defined on segments include "read", "write", and "execute". Segments may have a number of attributes, such as "date created", and "name of creator".

2.3 Protecting Object Types

This section reviews protection mechanisms found in contemporary operating systems. The access control list mechanism, to be provided as part of the case study VM, is described in greater detail.

The VM of this thesis provides a mechanism to express and enforce protection policies relating to objects. More precisely, the VM supports protection policies for segment objects and for all ETOs constructed (recursively) out of segment objects. Access to an object is expressed in a straightforward way: each operation on an object has an associated permission. The name of the permission is the same as the name of the operation. For example, there are "enqueue" and "dequeue" permissions for a message queue object, and "read", "write", and "execute" permissions for a segment object. A collection of access permissions for objects is called a domain [Lampson69]. A program executing "in a domain" is constrained to perform only those operations that are specified in the domain. Table 2-3 illustrates a domain comprising access permissions for three objects.

permission	object name
enqueue	MQ_23
enqueue, dequeue	MQ_31
read, execute	segment_16

Table 2-3

A Sample Domain

The domain model of protection is sufficient to characterize the protection facilities of most current general-purpose systems. In one realization of the domain model, a domain is represented as a set of capabilities [Dennis66]. A capability has two parts: a name of an object, and a set of access permissions. (1) In order that the protection policies not be circumvented, capabilities are tamperproof. Presentation of a capability specifying a given permission is a prerequisite for performing the corresponding operation on an object. Capabilities can be passed from one domain to another in order to achieve sharing.

In a capability system it is not necessary that the agent attempting to access an object be identifiable; mere possession of an appropriate capability indicates proper authorization. In another realization of the domain model, which makes use of access control lists [Saltzer75], each agent is identifiable, and has a globally recognized name. In access control list (ACL) systems, an agent that has the potential to access objects is called a principal. A principal may be characterized as the internal manifestation of authority in a computer system. Each principal is identified by a name known as a principal identifier. Every object has an associated access control list that contains ordered pairs of 1) access permissions and 2) principal identifiers. Every time that a principal attempts to reference an object, the associated access control list is searched to determine whether the type of access that the principal is attempting is allowed. (2) If it is not allowed,

---

(1) The name part of a capability may also contain type information.

(2) In this thesis we shall use the term "principal" informally, to replace the more precise phrase "processor executing a program on behalf of a principal".

the attempted reference is aborted. A sample access control list for a message queue object appears in Table 2-4.

permission	principal
enqueue	Smith
enqueue, dequeue	Jones
dequeue	Brown

Table 2-4

A Sample Access Control List

If the ACL shown in Table 2-4 were associated with "MQ_31", this would indicate that the principal named Jones could perform both enqueue and dequeue operations on "MQ_31".

The relative merits of the capability and ACL protection mechanisms have been described in the literature [Fabry74, Saltzer75]. The VM of this thesis provides for protection of objects (i.e. segments as well as ETOs) by ACLs. The primary motivation for choosing the ACL mechanism is that a working ACL-based system, namely the M.I.T. Multics system, is available for inspection and comparison by the author. An additional motivation is that the use of ACLs as a means of protecting extended type objects has not been explored in the research literature; current extensible systems use capability-based protection mechanisms. Providing for variable-length (and potentially large) ACLs is a design issue that is not encountered in capability-based VM subsystems.

2.4 Related Terminology and Assumptions

The purpose of this section is to introduce additional relevant terms, as well as some underlying assumptions. The terms introduced here are related to those defined in preceding sections.

We define a protected subsystem to be a set of programs and data that is encapsulated so that other programs may invoke those in the set only at specified entry points. Programs outside the set are prevented, by the encapsulation, from causing arbitrary transfers of control to or from encapsulated programs, and from directly accessing the encapsulated data. Each extended type manager in the case study VM is implemented as a protected subsystem.

We wish to allow for the case in which each protected subsystem may execute in a distinct domain. Accordingly, we specify a one-to-one correspondence between protected subsystems and domains. Given our informal use of the term "principal", each protected subsystem assumes the role of a principal. Consequently subsystems such as type managers that are not ordinarily associated with "end users" nonetheless have principal identifiers.

To summarize, we are assuming in this thesis that "principal", "protected subsystem", and "domain" can be used interchangeably. Further, we assume that each extended type manager is implemented as a distinct protected subsystem.

In general, this brief description of the nature of a domain, a principal, and related concepts will suffice as a basis for understanding this thesis. A more detailed description of the naming of these entities is not necessary here. We do assume, however, that each protected subsystem has a low-level name that has the form of a system-provided unique identifier. Such a name may be the same as the name of a canonical component, or it may be

associated with the protected subsystem as a whole. Possible representations for the principal identifier names that appear in ACLs include single unique identifiers or a sequence of unique identifiers. A further discussion of the naming of principals, domains, and protected subsystems appears in the work of Saltzer [Saltzer74] and Montgomery [Montgomery76].

Associated with each object type are two classes of principals. First, there is a class of principals that are consumers of objects of a given type. A consumer of a type is any principal that invokes the corresponding type manager. Second, there is a class of principals (with only one member) that are suppliers of a given object type. The supplier of an ETO is the corresponding ETM. Using the terminology introduced here, we can characterize each ETM as the supplier of one object type and the consumer of at least one object type.

In this thesis we do not assume any particular relationship between a protection context and an execution sequence. Thus, we wish to allow for the possibility that there can be 1) one-to-many, 2) one-to-one, or 3) many-to-one relationships between domains and virtual processors. (1) In particular, we generally will not specify the number of virtual processors that support a given ETM.

We do not consider the details of an interdomain communication mechanism in this thesis. However, we do assume that for any such mechanism, the unforgeable identity of the invoking principal is supplied to the target domain.

---

(1) A virtual processor may be either a real processor or an abstraction provided by multiplexing a real processor.

2.5 Authority Hierarchies

The authority structure represented in an access control list is not expected to be static. In this section we summarize one approach, to be supported by the case study VM, for implementing a hierarchical authority structure in an ACL-based system.

The operations to display and update the contents of ACLs must be controlled. Controls can be applied if each ACL has an ACL if its own. This second ACL might contain permissions for such operations as "display" and "update". (1) Controlling access to an ACL via another ACL suggests a hierarchy of access control lists. Rotenberg [Rotenberg74] describes a special protected subsystem, called an office, that can embody the access control policies that are likely to be expressed in such a hierarchy. Every ACL is under the control of exactly one office, which we shall call its controlling office. An office will determine, according to some internal policy, whether a given principal (or group of principals) can perform some operation on an ACL under its control. The escape mechanism provided by offices eliminates the need for a hierarchy of ACLs.

---

(1) We do not specify any controls on the operation of searching an ACL. Rather, we assume that any principal can search any ACL. Were this not the case, it would be necessary to determine, by searching the ACL of the given ACL, whether the given ACL could be searched. Clearly, in order that any reference to an object not result in an infinite loop of ACL searches, there must exist some ultimate ACL for which searching is unrestricted. We specify that the ultimate ACL be the same as the original ACL. For those cases in which the information in ACLs may be exploited as a covert information channel [Lampson73], non-discretionary controls [Saltzer75, Whitmore73] can be used to prevent a principal from obtaining such information.

2.6 Protecting Extended Type Objects in an ACL-Based System

The use of access control lists, rather than capabilities, for protecting extended type objects is novel. This section specifies the role of ACLs in an extensible system.

For every reference to an ETO, at least two ACLs are consulted. For example, suppose that Smith wishes to do an "enqueue" on MQ_26, and that the (only) component of MQ_26 is SEG_14. Upon being invoked, the ETM for message queues searches the ACL of MQ_26 to see if Smith has enqueue access. Assuming he does, the ETM searches its map for MQ_26 and finds SEG_14. To carry out the enqueue request, the ETM must write into SEG_14. It invokes the type manager for segment objects, which in turn searches the ACL of SEG_14 to see if the message queue manager has write access. In general, the height of the ETO representation tree is a lower bound on the number of distinct ACL searches.

Each extended type object corresponds to the root node of a representation tree. Any other object "X" in the tree must be a component of some object "Y". We observe that only the type manager for Y must be able to reference X. Therefore the only principal that should appear on the ACL of X is the principal for the type manager of Y. An ACL with only one principal is sufficient for every object in the tree except the root node object. We refer to this simplified form of ACL as a degenerate ACL. A degenerate ACL can be searched faster than a normal ACL; thus checking access to representation objects can be optimized.

In an ACL-based system, every ETM performs three basic mapping functions on the name of an argument object. First, it maps the object name to the object type, to be sure that the object passed to it is one of its own.

Second, it maps the object name to the associated ACL, to see if the consumer has appropriate access for the requested operation. Third, it maps the object name into the set of component objects, in order to carry out the requested operation.

2.7 ACLs Versus Capabilities in Extensible Systems

The effectiveness of ACLs and capabilities in extensible systems is contrasted in this section. An important difference between these two mechanisms is that the capability mechanism includes a means for exercising dynamic constraints on the invoked ETM. This is because, in a capability system, a type manager must derive its permission to access component objects from the capability that it inherits for the ETO. Two methods for obtaining access to component objects in a capability system are: 1) amplification of access rights [Jones73], and 2) unsealing a sealed capability [Redell74].

The amplification method is used in the Hydra system. In Hydra, the capabilities for the component objects of an ETO are stored in the "capability part" of the ETO. Consumers of an ETO have capabilities that express certain access privileges; however the capabilities of consumers do not contain privileges allowing them to load from or to store into the capability part of an ETO. When the supplier of an ETO is invoked, it obtains via amplification these "load" and "store" rights that are necessary for manipulating component objects.

The method of sealing and unsealing was pioneered in the CAL system, and is used in the SRI system. In the SRI system, each ETM is the sole possessor of a special capability -- called a "type" capability -- that permits it to perform "unseal" operations on certain other capabilities. Whenever an ETM is

passed a capability for one of its own ETOs, it can present that capability, together with its type capability, to a lower layer in the system and obtain all the component capabilities for the ETO in return. This operation is called "unsealing". Since other subsystems do not possess the particular type capability, they must regard the capability for the ETO as being "sealed".

Both the amplification and the sealing methods provide a finer degree of access control to component objects than an ACL-based system could provide. In the capability systems, ETMs can access the component objects only for the duration of their invocation. Cohen [Cohen75] calls this property conservation, and describes circumstances under which it may prove useful. In addition, Cohen shows how consumers in capability systems may further constrain the operation of ETMs by deliberately excluding certain rights from parameter capabilities. In an ACL-based system, such dynamic constraints could be enforced only if the interdomain communication mechanism were enhanced to allow consumers to express these constraints. Schroeder [Schroeder72] has proposed such enhancements for an interdomain communication mechanism. The VM of this thesis does not depend on these enhancements, but neither does it preclude them.

2.8 Directories as Extended Type Objects

Although this case study VM is strongly inspired by the Multics VM, one of the significant differences is that it supports dynamic type extension. In particular, directories are ETOs, as in the Hydra and SRI systems.

A directory object contains a variable number of entries, each of which contains a symbolic name and a machine-oriented name. Operations on a directory object include mapping symbolic names into machine-oriented names,

and adding and deleting entries. Since directories are objects themselves, the machine-oriented name of one directory may appear in another. Consequently, directories may be arranged in a tree-structured hierarchy.

The directories of this case study VM are implemented as extended type objects, with segments as the representation objects. Segments are a natural choice for the representation since, like directories, they are variable-length objects. Additional motivation for implementing directories as ETOs is provided in the work of Redell [Redell74] and Bratt [Bratt75].

2.9 Summary

The case study virtual memory subsystem of this thesis is based on the Multics, CAL, Hydra, and SRI virtual memory subsystems. As such, it should provide a nontrivial and practical context for the analysis of the next two chapters. The major objective of the next two chapters is to specify intermodule relationships in the VM. A secondary but necessary objective is to justify the particular modular decomposition of the VM, which is described in these chapters.

Chapter III

Treating Objects as Bindings

3.1 Introduction

In this chapter we apply the LISP notion of object bindings to the structuring of the case study virtual memory (VM) subsystem. To begin, we propose that the memory multiplexing function, which is fundamental to a VM implementation, should be provided at the lowest level. We then develop an object-oriented model of memory multiplexing, in which the objects of the model are related by bindings. The utility of this binding model is that, even though objects may be related by bindings, the respective object managers may be independent. Much of the chapter is devoted to a description of an implementation of the memory multiplexing model that preserves the independence of type managers inherent in the model. Certain dependencies, such as dependencies on an addressing environment, are introduced in an implementation, however. Finally we show that not one, but several, VM layers carry out a memory multiplexing function that can be characterized by the model. Consequently, observations about an implementation, and about dependencies, apply to a number of the layers in the case study VM.

3.2 Removing Unnecessary Dependencies in the Lower VM Layers

In the model for memory multiplexing the operations being carried out are merely the manipulations of bindings between objects. The operations on these objects are thus no more complex than the operations defined on LISP atomic objects. We show that there are type managers in the lower VM layers that need embody no knowledge of the semantics of certain other types. Rather,

these type managers simply store the names of objects of other types in their internal data bases. The binding and unbinding operations, which are sufficient to characterize memory multiplexing, are implemented as the storing and fetching of object names. This chapter concentrates on the parts of the VM that provide abstractions more primitive than segments, primarily because the application of the proposed techniques for eliminating classes of dependencies is most apparent in these lower VM layers. The VM abstraction provided by the VM layers of this chapter can be characterized as a potentially large number of spaces containing a potentially large number of potentially large objects.

3.3 Plan of the Chapter

The first section of this chapter justifies the choice of a simple memory multiplexing layer as the lowest layer in the VM. This layer provides the abstraction of a simple paged addressing environment. The implementation of the multiplexing function is modelled as a collection of type managers that can associate their own objects with others via explicit binding and unbinding operations. We show that these type managers can be partitioned into regions, so that the correct operation of one region does not depend on the correct operation of another. Next, we describe another layer of the VM that provides a memory abstraction that is like a primitive segment. Like the lowest layer, this second layer carries out a simple multiplexing function. Consequently it can also be structured as a collection of modules with few interdependencies. Finally, we describe a third layer that contributes to the VM abstraction by providing for a large number of spaces of primitive segment objects. The specification of this layer -- to carry out a simple multiplexing function --

is quite similar to that of the first and second layers. Since the specifications of the three layers are quite similar, the internal structures can be as well. Thus the structuring techniques that are based on the memory multiplexing model of this chapter can apply to each of these layers.

3.4 The Lowest Layer of the Case Study VM Subsystem

As mentioned in Chapter I, it is necessary to justify the particular modular structure of the case study VM. Since we will specify the case study VM in a bottom-up fashion, we first justify the functionality of the lowest layer. The lowest layer provides for the multiplexing of main memory.

In order to support a large number of objects -- whether they be segment objects or other abstract object types -- it is necessary to provide a way of multiplexing limited primary memory resources. The representation for an object is located somewhere in a hierarchy of storage devices, but during any short time interval only a subset of all the object representations is contained in primary memory. A memory multiplexing mechanism, by moving data between primary memory and the various storage devices, provides the illusion that all the representations are contained in primary memory. The memory multiplexing strategy can be justified as long as 1) computations exhibit locality of reference, 2) there is a spectrum of storage devices that comprises large, slow, inexpensive-per-bit devices at one end and small, fast, costly-per-bit devices at the other; and 3) the cost of moving information from one part of the hierarchy to another is relatively low.

We are claiming that memory multiplexing is necessary to support a large number of objects. In addition, it may be necessary to multiplex main memory among different parts of the representation of any single object, if that

object is large. For these reasons we consider a VM model with a simple memory multiplexing facility at the lowest layer.

3.5 Overview of Memory Multiplexing

The abstraction provided by multiplexing can be described in terms of several sets of objects. The first set is a large set of state objects, and the second set is a small set of operational objects. Multiplexing is a means for simulating a third set such that: 1) the size of the third set equals the size of the first set, and 2) the elements of the third set are operational objects like those in the second set.

In a VM implementation, the first set corresponds to addressable sections of secondary memory and the second set corresponds to addressable sections of primary memory. In this thesis we refer to the first set as the set of home objects and to the second set as the set of frame objects. The set of home objects is assumed to be larger; i.e. we assume the amount of secondary memory is greater than the amount of primary memory. However, only the frame objects are operational objects -- only they can be referenced directly under program control without any real-time delays. The frames are multiplexed among the homes to produce a third set of abstract information containers. We refer to this third set as the set of block objects. By assumption the home, frame, and block objects are all the same size; i.e. they contain the same number of bits. In this pure memory multiplexing model, there is a one-to-one correspondence between block objects and home objects. (1)

---

(1) This pure memory multiplexing model is arbitrarily restricted to a two-level memory architecture. The model could be extended to accommodate a multilevel memory architecture if the higher level of every two devices were regarded as primary memory and the lower regarded as secondary memory. However, explicit consideration of multilevel memories is beyond the scope of this thesis.

3.6 A Model of Memory Multiplexing

We now examine the objects in the memory multiplexing model in detail. Like LISP atomic objects, these objects in the model have bindings, and operations to manipulate these bindings.

To complete the description of memory multiplexing, we need to introduce a fourth kind of object called a data object. A data object is a fixed-sized collection of bits -- the same size as home, frame, and block objects. It is strictly an abstract construct, so there is no corresponding physical representation. The concept of a data object is necessary for our object-oriented description of memory multiplexing. The name of a data object is equal to its contents. Thus, if data objects contain K bits, then there are 2^K data objects, each with a distinct K-bit name. Data objects have no bindings, and there are no operations defined on them.

The home object is an abstraction of an addressable section of secondary memory. A home object has a name and a binding. The name of a home object corresponds to a secondary memory address. For example, in a system that uses disks to provide secondary memory, the name of a home object might correspond to the union of: 1) a controller number, 2) a device number, 3) a cylinder number, 4) a track number, and 5) a record number. The binding of a home object designates a data object. Every home object is bound to some data object; in general more than one home object may be bound to the same data object. Since the (only) binding of a home object designates a data object, the binding is called the data binding of the home object. There are two operations on a home object, called the fetch and store operations. However, these operations are defined jointly on home and frame objects, so they will be described together with the other operations on frame objects.

The frame object is an abstraction of an addressable section of primary memory. A frame object has a name and two bindings. The name of a frame object corresponds to a primary memory address. In an implementation the frame name would be the absolute address of the first addressable unit (e.g. the first word or byte) of the section of primary memory. A frame object has a data binding and a home binding. The data binding designates a data object. More than one frame object may be bound to the same data object. The home binding designates either a home object or a special object called NULL. We impose a restriction on home bindings of frames that is necessary for the correct behavior of this memory multiplexing model: non-null home bindings must designate distinct home objects. Examples of binding relationships are depicted in Figure 3-1. The data, home, and frame objects in this figure are the atomic objects of the memory multiplexing model. A reference to the information "in" a frame or home object is an informal way of referring to the data object designated by the data binding of the frame or home object.

The operations defined on frame objects are

1) assign (frame_name, home_name),
2) release (frame_name),
3) read (frame_name, data_name),
4) write (frame_name, data_name),
5) fetch (frame_name), and
6) store (frame_name).

The assign operation sets the home binding of "frame_name" to "home_name". It enforces the constraint mentioned above, that no two frame objects may have the same home binding. The release operation sets the home binding of "frame_name" to NULL. The read operation returns "data_name", the current

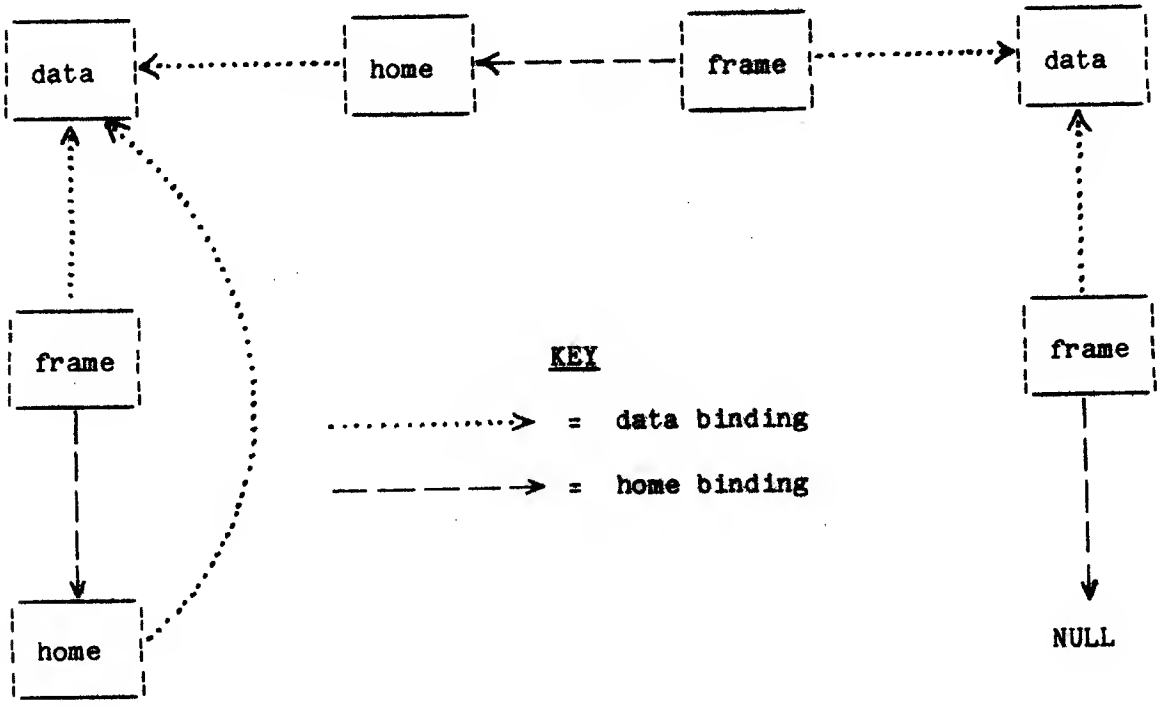


Figure 3-1

Bindings Between Objects of the Multiplexing Model

data binding of "frame_name", while the write operation sets the data binding of "frame_name" to "data_name". The effect of each instruction in a processor instruction set on the objects of the model can be represented as a combination of read and write operations. The fetch and store operations correspond to the I/O operations of fetching information from secondary memory into primary memory, and storing information from primary memory into secondary memory, respectively. Neither the fetch nor the store operation requires a home name as an argument. The appropriate home binding is implicit, since it must have been set by an assign operation before any fetch

or store could take place. The fetch operation replaces the data binding of "frame_name" by the data binding of the implied home name. The store operation replaces the data binding of the implied home name by that of "frame_name".

The fetch and store operations are defined jointly on home and frame objects. Each operation requires access to the data binding of both a home object and a frame object. The assign operation, in contrast, is defined only on a frame object. It requires the name of a home object as a parameter, but it places no interpretation on this name. The fetch and store operations could be decomposed further; e.g. the fetch operation involves: 1) obtaining the data binding of a home object and 2) setting the data binding of a frame object. However, treating fetch and store as indivisible operations more closely reflects the behavior of actual I/O commands.

3.7 The Block Abstraction

Up to this point we have described the objects that serve as component objects for the block object. The subsystem that acts as the type manager for block objects, which we shall call the block layer, manipulates the bindings of these component objects to produce the block object. This section provides a specification of block objects.

As we have mentioned, in the pure memory multiplexing model there is a one-to-one correspondence between blocks and homes. For example, the set of home names could also serve as the set of block names in a pure memory multiplexing scheme. However, if a one-to-one correspondence exists, the set of block names will be as large as the set of home names, and individual block names will be quite long. If block names were short (on the order of ten

bits) rather than long (on the order of twenty bits) then the main memory requirements for programs that manipulate block objects could be significantly reduced. To economize on main memory usage, we abandon the one-to-one correspondence of the pure memory multiplexing model and specify that the set of block objects be smaller than the set of home objects. In this case, we reference block objects with shorter names (called local machine-oriented names by Bratt [Bratt75]) and provide operations to associate block and home objects. The cost of periodically re-establishing associations between block and home objects is acceptable, assuming that programs using block objects exhibit locality of reference. (1) Unlike the pure memory multiplexing model, the model we have chosen involves two kinds of multiplexing. First, there is multiplexing of main memory: the small set of frames is multiplexed among the large set of homes. Second, there is multiplexing of the block name space: the small set of block names is multiplexed among the large set of home names.

Block objects have two bindings that are visible above the block layer interface: a home binding and a data binding. The home binding may designate a home object or NULL. To allow sharing, more than one block object may have the same home binding. Thus there is no restriction on the home binding, as there is in the case of frames. A consumer of a block object may invoke the block layer to set the home binding of a block. Thereafter, a consumer can reference (i.e. read and write) the block directly. Each reference to a block object either returns or changes the data binding. Each block object has one hidden binding, called the frame binding. The term "hidden" is used here as

---

(1) The general strategy of assigning short, temporary local names to objects that already have long, permanent global names is employed in many general-purpose operating systems, including the Multics, Hydra, and CAL systems.

Robinson has defined it [Robinson75], meaning that consumers of block objects cannot determine the existence of the frame binding. The frame binding designates either a frame object or NULL.

The block layer provides not just one, but rather several, spaces of block objects. These spaces of block objects, called block spaces, form the addressing environments for various subsystems in the system supervisor. (1) In particular, some subsystems in the VM execute in a block space addressing environment. Other subsystems in the case study VM rely on more sophisticated memory abstractions such as segmentation.

There are four visible operations defined on block objects. These are

1) initiate (block_name, home_name),
2) terminate (block_name),
3) read (block_name, data_name), and
4) write (block_name, data_name).

These four operations, unlike all the operations defined on frames and homes, can be invoked by programs outside of the block layer. In the case of each of these operations, the particular block space is an implicit additional argument. The initiate operation replaces the home binding of "block_name" by "home_name". The terminate operation replaces the home binding of "block_name" by NULL. A principal can therefore control which homes are bound to blocks in its block space by means of the initiate and terminate operations. The read and write operations are analogous to the read and write operations on frames. Any instruction referencing a block (e.g. "exclusive OR

---

(1) A "level one processor" in the two-level implementation of virtual processors described by Reed [Reed76] is an example of a subsystem that may make use of a block space as an addressing environment.

to memory") is implemented in terms of the read and write operations. The two operations for manipulating the (hidden) frame binding of a block are

1) connect (block_name, frame_name), and

2) disconnect (block_name).

The connect operation sets the frame binding of "block_name" to "frame_name". The disconnect operation sets the frame binding of "block_name" to NULL. These two operations can be invoked only by programs in the block layer.

Any processor instruction that references a block object does so using a two-part address of the form

(block_name, offset).

The offset is typically given in units of words or bytes. The value of the offset is irrelevant in this memory multiplexing model; all write operations on a block change the data binding and all read operations leave it unchanged. The block space addressing environment is similar to that provided by the TENEX system [Bobrow72]. (1)

The objects of the memory multiplexing model, and the operations on them, have now been defined. Each of the operations, except fetch and store, involves only the manipulation of a binding designating some target object. In verifying the correct operation of a module that manipulates a binding, the

---

(1) In TENEX the blocks, which are 512 words in size, are considered to be concatenated so that they form a single linear space. To reference

(blockname, offset)

in TENEX, a single address with the value

blockname * 512 + offset

is presented.

semantics of the target object are irrelevant. There is no strong dependency on the module managing the target object.

3.8 Overview of Block Layer Implementation

We now consider how the bindings of home, frame, and block objects might be implemented. It is the intent in this section, as well as in following sections that describe some aspects of the implementation in greater detail, to show that the modular independence inherent in the multiplexing model can be preserved in an implementation.

The representation of any data binding is implicit, in the sense that the data binding of an object *is* the contents of the object. The remaining bindings, on the other hand, are implemented in distinct data structures. One straightforward way of implementing these bindings is to use tables such as those shown in Figure 3-2. The *block space table* of Figure 3-2a represents

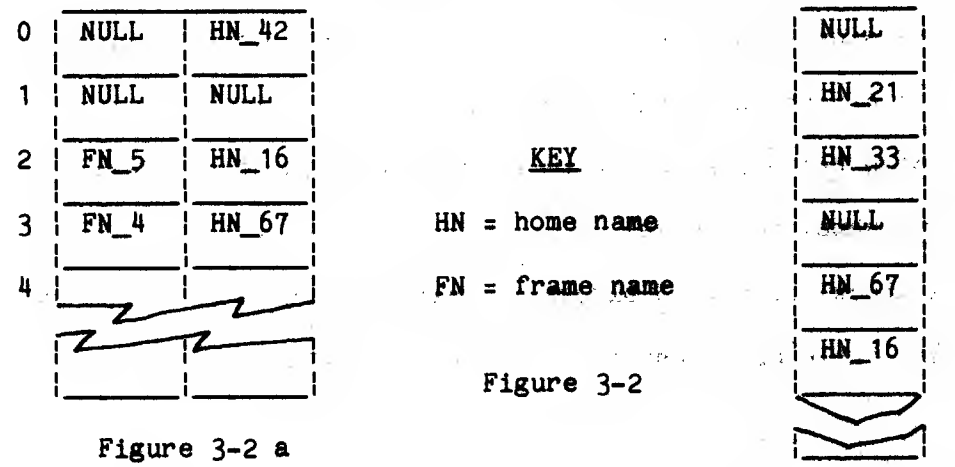


Figure 3-2

Figure 3-2 a

Figure 3-2 b

Tables Representing Frame and Home Bindings

the home and frame bindings of each block in a block space. The framelist table of Figure 3-2b represents the home binding of each frame in primary memory. There is one framelist, but there are as many block space tables as block spaces. Because these two kinds of tables support distinct object types, they are managed by distinct subsystems within the block layer. The subsystem that manages the bindings of block objects, by manipulating entries in block space tables, is called the block sublayer. The subsystem that manages the bindings of frame objects, by manipulating entries in the framelist, is called the frame sublayer. The frame sublayer performs fetch and store operations, so it manages data bindings of home objects as well. Block layer programs can reference these tables as block objects, since the addressing environment of the block layer (to be described in the next section) is a restricted form of block space.

We can now describe the effects of the initiate, terminate, assign, release, connect, and disconnect operations on the data bases of the block layer. It is not necessary to consider the effects of read, write, fetch, or store operations since these operations reference only data bindings, and therefore do not affect the data bases of Figure 3-2.

A block space table that contains N ordered pairs of the form

<frame name, home name>

can describe a block space of N blocks. Initially, many of the ordered pairs in a block space table may consist of two null components. (1)

---

(1) Even in a "new" block space, however, some blocks will be associated with common utility procedures and procedures that serve as toeholds to the block layer. Similarly, the distributed supervisor of the Multics system appears in every newly-created Multics address space.

Initiate operations set the home name parts of the ordered pairs;

initiate (3, home_name_67)

sets the home name part of the 3rd ordered pair in the block space of the caller to "home_name_67"; i.e. the home binding of block 3 is now the name "home_name_67", as shown in Figure 3-2. The frame name part of the ordered pair is set by the connect operation. Performing the operation

connect (3, frame_name_4),

sets the frame name part of the 3rd ordered pair in the block space table to frame_name_4, as shown. Entries in the framelist are set by the assign operation, so performing the operation

assign (frame_name_4, home_name_67)

sets the 4th entry in the framelist to home_name_67, as shown in Figure 3-2. The respective inverse operations -- terminate, disconnect, and unassign -- set values of bindings to NULL.

As mentioned, block space tables are implemented as blocks and therefore have underlying frames. The frame name of the block space table associated with an executing principal is stored in a special processor register. Virtual addresses of the form

(block_name, offset)

are converted to absolute addresses by the simple calculation

address = frame_name_part (table + W*block_name) + offset;

in which "table" denotes the frame name of the block space table and W denotes the number of words occupied by each entry in the block space table. The value of "address" is simply storage location number "offset" in the desired frame name. Thus read and write operations on blocks are mapped into read and write operations on frames.

3.9 The Addressing Environment of the Block Layer

As mentioned above, the block layer uses block spaces for its addressing environment. The procedures and data bases of the block layer are addressed as blocks. The particular block spaces used by the block layer are called basic block spaces.

The addressing environment of the basic block space is preset; programs executing in the basic block space do not perform initiate or terminate operations. Furthermore, the frame binding of every block is guaranteed to be fixed and non-null. Since blocks in the basic block space have fixed, non-null frame bindings, they do not need (or have) non-null home bindings. Only the data bindings of the blocks (and underlying frames) in a basic block space can change. As a consequence, the block layer never need be invoked to manipulate its own addressing environment. Unlike the block space abstraction that the block layer provides, the basic block space that it uses is completely static. The motivation for interpreting this static environment as a set of blocks is merely to achieve economy of hardware (or firmware) mechanism. Just as there are a number of (real) block spaces available to support supervisor subsystems, so are there several basic block spaces available in order to isolate functional components of the block layer.

The basic block space addressing environment is essentially the same as the environment provided by "level 0" in Parnas' family of operating systems. It is also quite similar to the environment provided by Multics "unpaged segments". (1)

---

(1) Multics unpaged segments are not of uniform length. Rather, the length is dictated by function, to conserve primary memory. Blocks in a basic block space may also be allowed to be different lengths, without complicating the supporting mechanism.

3.10 Handling Frame Faults

In preceding sections we have described the objects of the memory multiplexing model, and suggested an underlying implementation. The block sublayer implements the bindings of block objects, and the frame sublayer implements the bindings of frame and home objects. In addition to these subsystems that serve as type managers for objects of the multiplexing model, there are additional block layer subsystems that invoke these type managers. These subsystems, which allocate frames on demand to support consumers of block objects, are described in this section.

If a principal references a block with a null frame binding (but with a defined home binding) then a frame fault occurs. In this thesis we choose to model the handling of a frame fault by two cooperating principals: a frame-claiming principal and a frame-freeing principal. Each of these principals executes in a basic block space. The frame claimer provides the faulted block with a supporting frame. The frame freer wrests frames away from blocks, in an orderly manner. (1)

Providing distinct principals that handle frame faults is consistent with the principle of least privilege [Saltzer74], since handling a frame fault does not require access to programs or data of the faulting principal. Furthermore, if the frame claimer and frame freer are implemented as loosely-coupled virtual processors, system performance may improve as a result of their parallel activity. Other advantages of implementing these two VM mechanisms in distinct virtual processors are given by Reed [Reed76].

---

(1) An experimental version of the Multics paging software developed by Huber [Huber76] makes use of a dedicated virtual processor for the frame freer.

The frame claimer and frame freer are described briefly. The intent here is not to show algorithms in detail, but rather to indicate how these two principals invoke the block and frame sublayers to manipulate the bindings of block, frame, and home objects. Although multiple frame claimers and freers could exist, this description is sufficient only for a single frame claimer and a single frame freer.

An explanation of some terms and notation is necessary, before listing the steps followed by the frame claimer and frame freer. Each step in the frame claimer and in the frame freer corresponds to one invocation of either the block or frame sublayer. The steps below that invoke the block sublayer begin with "[B]", and those that invoke the frame sublayer begin with "[F]". Some of the steps correspond to utility functions provided by the block or frame sublayers. The remaining steps correspond to binding manipulation operations, such as the "connect" operation, that have been described previously.

A frame is called a free frame if its home binding is null. The list of free frames is implemented as a thread running through the framelist. Associated with each frame is a list of ordered pairs called a trailer list. The first component of each pair designates a block space table, or equivalently, a block space. The first component can be implemented as the frame name of the block space table. The second component is a block number, or block name, in the given block space. A trailer list indicates those blocks (in respective block spaces) that are bound to a frame.

First we sketch the operation of the frame claimer. Any principal that takes a frame fault invokes the frame claimer. The arguments passed to the frame claimer are a (block space, block name) pair. The frame claimer carries out the following steps.

1. [B] Obtain the home binding of the (block space, block name) pair, i.e. get a home name.
2. [F] Determine if there is any frame that is bound to this home. If so, get the name of this frame and go to step 7.
3. [F] Get the number of free frames. If this number is less than a certain threshold value, signal the frame freer. If there are no free frames, wait for a signal from the frame freer.
4. [F] Select a free frame.
5. [F] Assign the home name of step 1 to the frame name of step 4. This decrements the number of free frames.
6. [F] Perform a fetch operation on the chosen frame name. This updates the data binding of the frame.
7. [F] Add the (block space, block name) pair to the trailer list of the chosen frame.
8. [B] Perform a connect operation, which sets the frame binding of the block in the (block space, block name) pair to the chosen frame.

The frame freer is activated whenever the frame claimer detects that more frames should be freed. The frame freer carries out the following steps.

1. [F] Get the number of free frames. If the number is greater than zero, signal the frame claimer. If the number is greater than the threshold value, wait for a signal from the frame claimer.
2. [F] Select a frame that is not free. At least one block must be bound to this frame.
3. [F] Find a (block space, block name) pair in the trailer list of the chosen frame.
4. [B] Disconnect the chosen frame from the block found in step 3.
5. [F] Remove the (block space, block name) pair found in step 3 from the trailer list of the chosen frame.
6. [F] If the trailer list is not empty, go to step 3.
7. [F] Perform a store operation on the chosen frame name. This updates the data binding of the associated home.
8. [F] Perform a release operation on the chosen frame. This increments the number of free frames. Go to step 1.

The binding states that correspond to steps in the claiming and freeing sequences are shown in Figure 3-3. Initially, a given block, frame, home, and data object are in state A. In state A, the various bindings are as they should be following an initiate operation. The frame claimer operates on a collection of objects that is in state A. Steps 5, 6, and 8 of the frame claimer cause state transitions to states B, C, and D respectively. The respective operations performed in these steps are assign, fetch, and connect. The frame freer operates on a collection of objects that is in state D. Steps 4, 7, and 8 of the frame freer cause state transitions to states C, B, and A

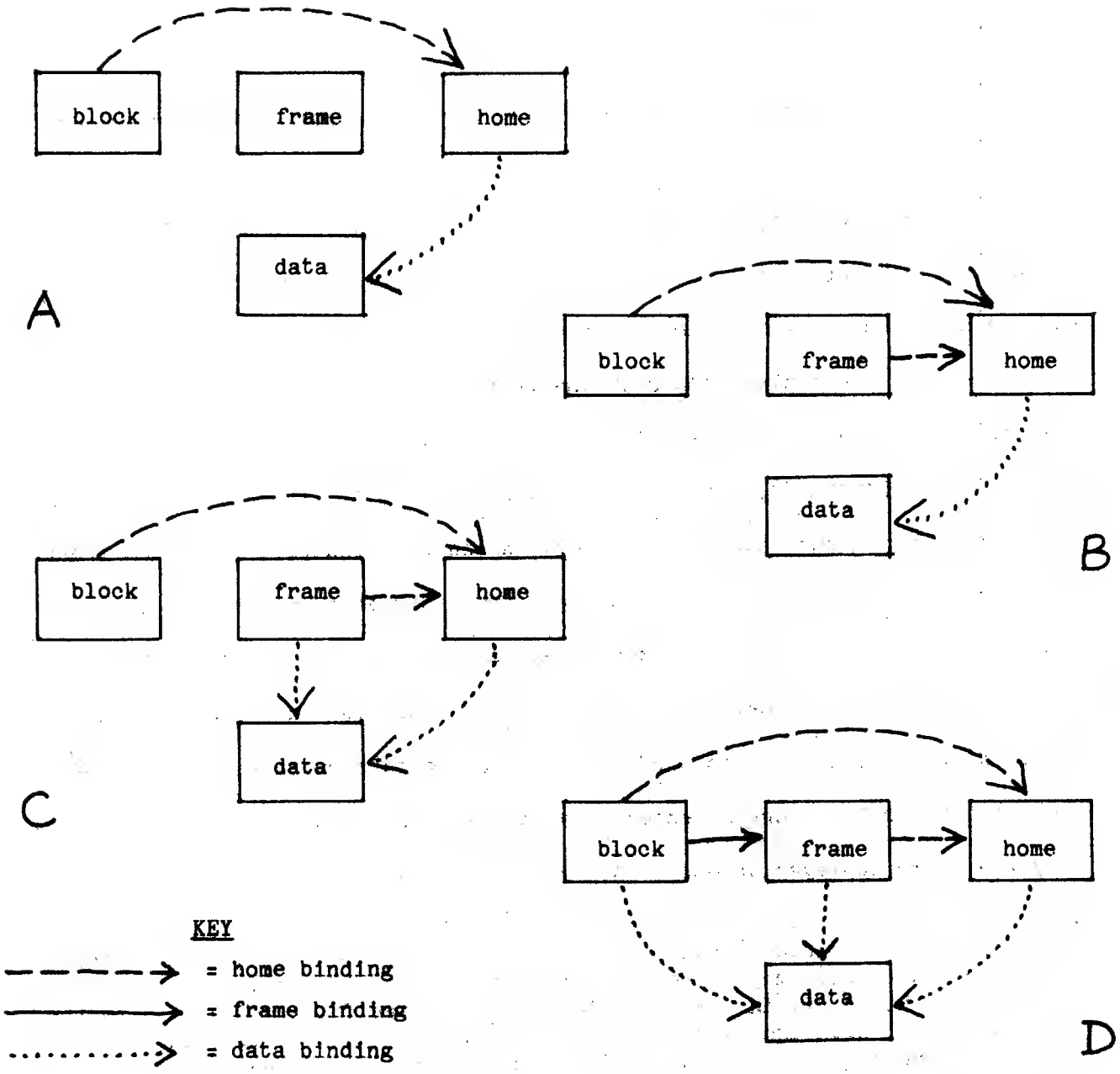


Figure 3-3

Binding States During Frame Claiming and Freeing

respectively. In this case, the respective operations are disconnect, store, and release. (1)

Of the sixteen steps listed above, six correspond to binding manipulation operations. The other ten are utility functions provided by the frame or home sublayers. These utility functions require their own supporting data structures. For example, step 2 of the frame claimer must locate the frame that is bound to a given home. It can do so efficiently by searching a balanced tree [Knuth73], with as many nodes as there are frames, in which each node maps a home name into a frame name. As another example, the frame freer must have an efficient way of unbinding a frame object from a set of block objects. The trailer list provides an efficient way of locating the blocks that are bound to a frame. (2) The trailer list provides an efficient way of locating these block objects. Of course, these data structures that support the utility functions are accessed only by the appropriate type managers; e.g. only the frame sublayer accesses the trailer list.

---

(1) If a block in a block space has a non-null frame binding, then the home binding of that block may be found either 1) in the block space table entry for that block, or 2) in the framelist entry for the frame designated by the frame binding. Although this redundant information can be tolerated in a VM model, it cannot be in a practical implementation, since it wastes memory space. In an implementation, a block space table entry would contain either a home name or a frame name. The connect operation would replace the home name by a frame name, and the disconnect operation would replace the frame name by its home binding, to be found in the framelist.

(2) This method of recycling frames, by removing their names from block space tables, is similar to the method used in Multics. Another strategy for managing the bindings between blocks and frames would be to put unique identifiers, rather than frame names, in the block space tables. These unique identifiers could be mapped into frame names using a central, hardware-supported associative memory. In this case a frame could be unbound from a set of block objects merely by deleting the corresponding (unique identifier, frame name) pair from the associative memory. Building an associative memory of the required size and speed seems within the state of the art.

3.11 Dependencies of Regions Within the Block Layer

In the preceding sections we described four subsystems that are part of the block layer: the block and frame sublayer and the frame claimer and freer. In this section we examine the interdependencies among these four subsystems, and show that only a few strong dependencies exist.

The strong dependencies that do exist include dependencies on an addressing environment. Each of the four subsystems executes in a basic block space. Since, as mentioned before, the basic block space environment is quite static, only the data bindings of the supporting objects ever need to be changed. Other bindings do not change; for example, a block in a basic block space is permanently bound to a particular frame. Only the read and write operations on blocks, provided by the block sublayer, and the read and write operations on frames, provided by the frame sublayer, are needed to support a basic block space. Thus any principal that executes in a basic block space strongly depends on the parts of the block and frame sublayers that support the read and write operations.

These parts of the block and frame sublayers, on the other hand, do not depend on any other parts of the block or frame sublayers. The read and write operations for frames merely manipulate data bindings of frames. The read and write operations for blocks map block names into frame names. Although the mapping relies on parameters from block space tables, these block space tables are referenced only to effect read and write operations for blocks in a basic block space. The remaining parts of the block layer never reference these block space tables since the basic block space environment is a static one.

In summary, each subsystem in the block layer depends on the mechanisms that provide read and write operations for block and frame objects. The

correct operation of these mechanisms, on the other hand, does not depend on any other block layer facility. It then seems reasonable to separate these read and write mechanisms from their respective block and frame sublayers, since they represent a greatest common mechanism. Since these mechanisms would be implemented in hardware in a practical system, they would be protected from interference by block sublayer and frame sublayer programs.

We now consider whether the frame freer and frame claimer depend strongly on the block and frame sublayers. They certainly depend weakly on the block and frame sublayers, since their rate of progress is controlled by the values of arguments returned by those sublayers. The interface to the block and frame sublayers can be specified in such a way that their erroneous operation would only delay the progress of the frame claimer and freer. Designing the interface in this way can be useful; for example, the frame claimer and frame freer could be implemented, and then tested using dummy arguments, before the block and frame sublayers were implemented. If our specification of the frame claimer and freer were that they manipulated uninterpreted arguments, then they would not depend strongly on the block and frame sublayers. However we specify that the frame claimer must coerce frame objects from the free state to the claimed state, and the frame freer must effect the reverse transition. The frame claimer and freer cannot satisfy this specification unless the underlying block and frame sublayers operate correctly. Hence, the frame claimer and freer depend strongly on the block and frame sublayers.

It may seem tempting to specify that subsystems such as the frame claimer and frame freer simply manipulate uninterpreted arguments. The block and frame sublayers have been given such a specification. As one moves up in a hierarchy of abstract machines, however, what appears to be uninterpreted data

at a given layer can and generally should be considered to be an abstract type at a higher layer. If many modules in a complex system are insensitive to errant behavior, it becomes extremely difficult to isolate the erring modules. We emphasize that we are not attempting to eliminate every intermodule dependency; this is neither possible nor desirable. We are trying to eliminate unnecessary dependencies, and loop dependencies in particular.

With the exception of the strong dependencies just described, there need be no other strong dependencies between subsystems in the block layer. The block sublayer does not depend on the frame sublayer, and vice-versa, since neither interprets bindings to objects managed by the other. Even the utility functions that are performed, such as the traversal of a trailer list, are carried out without interpretation of object bindings. In addition, the frame claimer and frame freer do not depend strongly on each other, since their only interaction is to synchronize their progress.

The block layer itself is strongly dependent on all of its components. The block layer depends on the block sublayer to maintain the proper correspondence between a block object and a frame object. If the block sublayer returned some other frame name, the data binding of a block object would be affected in a way that would not correspond to the specification of read and write operations for blocks. In particular, a read operation on a block might, in violation of the specification, cause the data binding of the block to change. Similarly, the frame sublayer must always choose the home name corresponding to a given frame name when undertaking a fetch or store operation. If it chose some other home name, the data binding of a block object could be changed even though a write operation had not been performed. The block layer is strongly dependent on the frame claimer and frame freer as

well. By operating incorrectly, either of these two subsystems in the block layer could change the value of an object binding. Such an erroneous modification would prevent the block layer from meeting its specification. The dependencies among block layer modules are illustrated in Figure 3-4.

3.12 The Next Layer in the Virtual Memory

The block layer provides not only a primitive virtual memory for the layers above it, but also the heart of the mechanism needed to implement the segment object. In this section we describe another region in the case study VM that is necessary for the support of segments.

The block space abstraction can be characterized as a small number of spaces containing a large number of small objects. In contrast, the abstraction specified at the beginning of this a chapter is a potentially large number of spaces containing a potentially large number of potentially large objects. To extend the block space abstraction to the desired abstraction, there must exist facilities for: 1) making large spaces (or objects) out of small ones, and 2) growing and shrinking the size of spaces (or objects). The first facility is provided by the block layer subsystem. We now describe the second facility. It provides for the allocation and freeing of home objects. Allocation and freeing of home objects is an economic necessity since the number of home objects is finite. We refer to the layer that performs these functions as the home allocation layer or home allocator. The two operations provided by this layer are

allocate (home_name), and

free (home_name).

The home name is an output argument in the first operation and an input

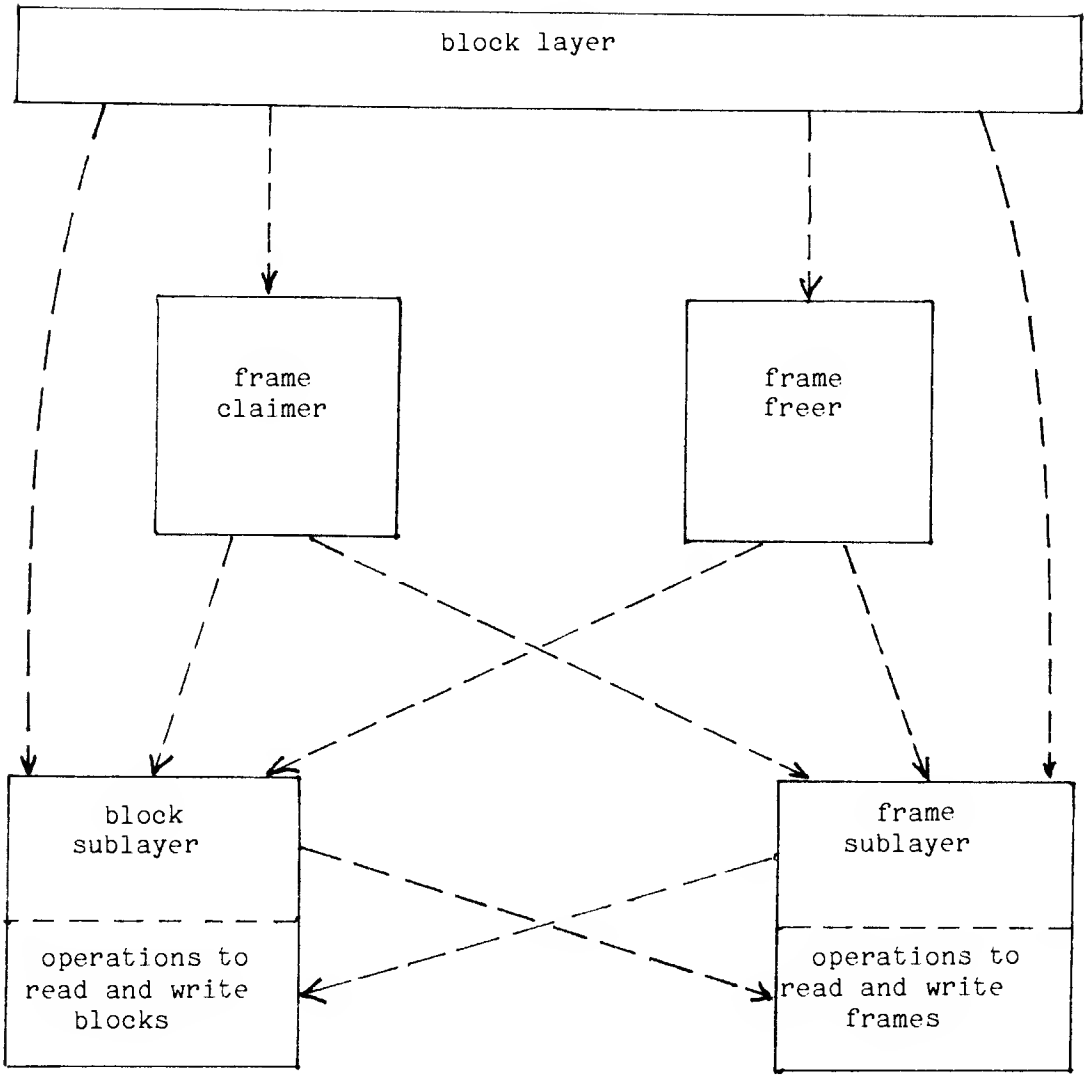


Figure 3-4

Dependencies Among Block Layer Modules

argument in the second operation. The specification of the home allocator is quite simple: to change the state of "home_name" from free to allocated, or vice-versa, on any invocation.

The home allocator maintains a data base that indicates whether or not any home object is allocated. This data base, called the homelist, may in general be large enough that only portions of it occupy primary memory at any time. (1) A natural way to multiplex primary memory among portions of the homelist is to implement the homelist as a set of block objects. The home allocator is thus a consumer of the abstraction provided by the block layer.

3.13 Dependencies Between the Block and Home Allocation Layers

As mentioned before, the addressing environment of the block layer is static. The block layer does not need to request the allocation or freeing of homes. Hence it does not depend on the home allocator. The home allocator, which is a consumer of block objects, does depend on the block layer.

To see how the home allocator might depend on the block layer, we need to consider: 1) how the home allocator uses block objects, and 2) the failure modes of the block layer. Since the blocks forming the homelist are supplied by the block layer, and since the block layer might fail to manage object bindings properly, the homelist could become garbled. Consequently the "allocated" attribute of a home may no longer be correctly represented. The home allocator might therefore allocate the same home object twice, without any intervening "free" operation. Since the home allocator programs could not detect a garbled homelist (unless redundant information were kept in the

---

(1) The M.I.T. Multics data base that corresponds to the homelist currently is about 500,000 bits in size.

homelist or another data base), they could continue to operate unperturbed. In a narrow sense, the home allocator would be operating "correctly" since a failure of the block layer would not cause it to "blow up". However, a failure of the block layer can cause the specification of the home allocator to be violated. According to our definition of (strong) dependency, the home allocator thus depends upon the block layer.

3.14 Specification of Large Block Objects

The block layer together with the home allocation layer produce an abstraction that can be characterized as a small number of spaces containing a potentially large number of small objects. We now extend this abstraction, to one that can be characterized as a small number of spaces containing a potentially large number of potentially large objects. These objects, called large blocks, are addressed just like block objects, i.e. via 2-part addresses. The maximum length of a large block, however, is much greater than the small, fixed length of a block object. Since large blocks are to be variable-length objects, we specify an operation,

set_length (large_block_name, length),

that can grow or shrink the current length of a large block. The initial state of a large block (which includes a current length) is specified in an initializing operation

initialize (large_block_name, initial_attributes).

By definition the initial contents of any part of the large block between the beginning and the current length is zero. Reading or writing a large block at a point beyond the current length causes an error.

3.15 Implementation of Large Blocks

Each large block can be represented by a table called a large block table, (LB table) which contains M ordered pairs of the form

<frame name, home name>.

The home named in the i-th pair contains the data of the i-th piece of the large block, and the frame named in the i-th pair designates the current primary memory frame (if any) for the data. A large block table may contain, in addition to the ordered pairs, certain attributes of the corresponding large block object.

Although there is a considerable difference between the block and the large block abstractions, there is a significant common mechanism for supporting them. Many of the operations that need to be performed on a large block table to support large blocks are the same as those that must be performed on a block table to support blocks. For example, 1) adding a new block to a block space and 2) adding to the amount of (nonzero) information in a large block can both be supported by "initiate" operations that change the appropriate underlying tables.

Since there may be numerous large block tables, it is a design objective that only a subset of them need be in primary memory at any time. A natural way to achieve this objective is to implement each large block table as a block object. Thus, a large block space is realized as a space of blocks, each of which contains a large block table. Each large block table, in turn, describes a collection of homes (and possibly frames) that form the large block. In the context of large block space implementation, we shall refer to any block space table that describes a space of large block tables as a large block space (LBS) table. These data structures are shown in Figure 3-5.

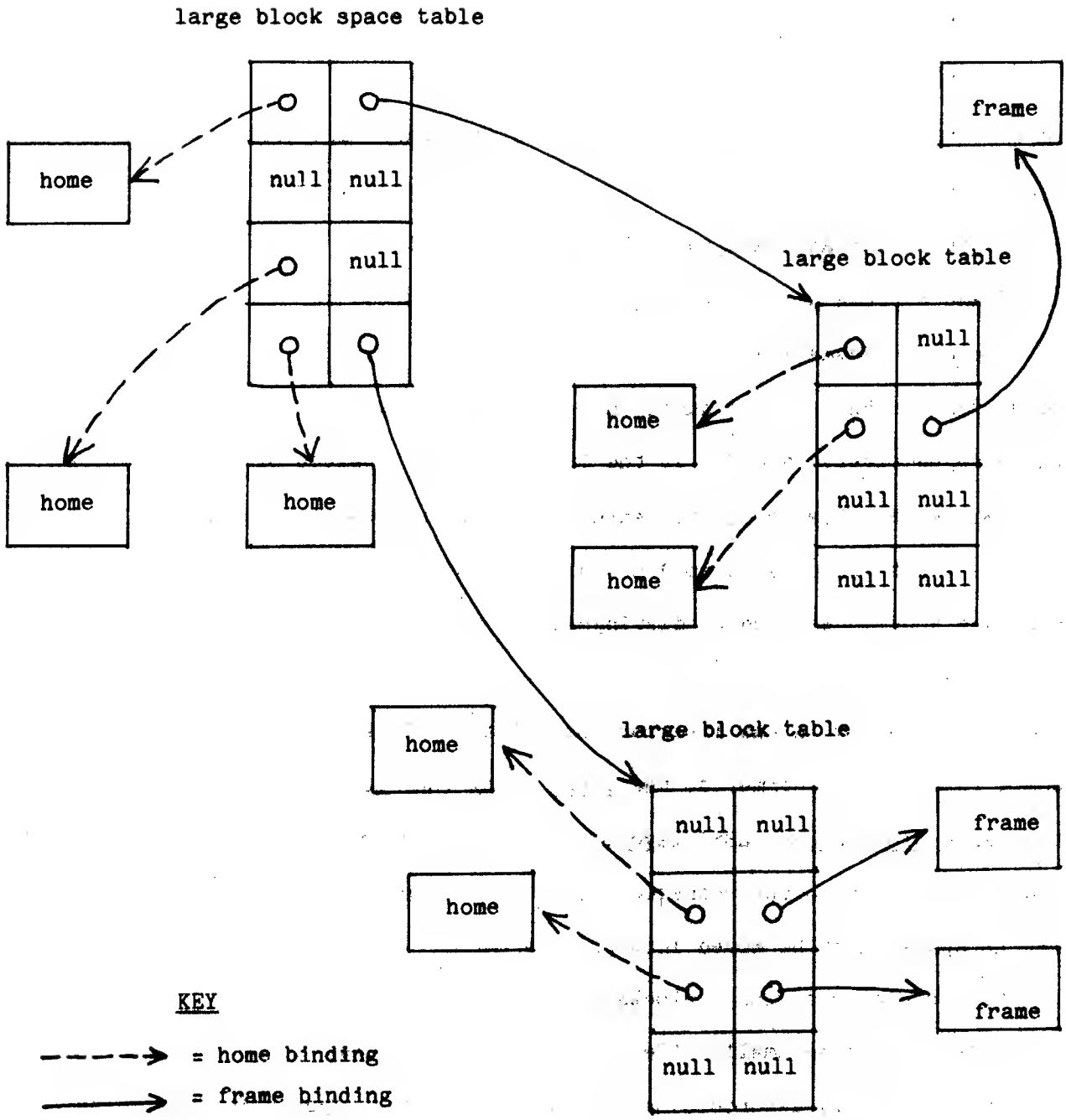


Figure 3-5

Data Structures Supporting a Large Block Space

Virtual addresses of the form

(L , O),

in which L is a large block name and O is an offset, are translated by the hardware as follows.

1. LB_table = frame_name_part (LBS_table + V * L)
2. frame = frame_name_part (LB_table + W * [O/F])
3. address = frame + MOD (O,F)

Here, "V" and "W" are the number of words per entry in an LBS table and an LB table respectively, and "F" is the frame size for a piece of a large block.

Since LB tables are implemented as block objects, it is possible to take a frame fault either in step 1 or in step 2 of the virtual address translation sequence above. We wish to distinguish between the incarnations of the block layer that deal with leaf nodes of the large block implementation tree and those that deal with the interior nodes (large block tables) of the tree. We shall refer to the former as the large block (LB) layer and to the latter as the large block space (LBS) layer. The LBS layer is responsible for handling frame faults that occur whenever any LB table, representing one large block in the LB space, must be moved into primary memory. The LB layer handles frame faults for pieces of large blocks. (1)

There are some small differences between the operation of the LBS layer and the previously described operation of the block layer. In particular, the frame freer of the LBS layer cannot arbitrarily free frames that contain LB tables. To see this, we refer to the example in Figure 3-6. For clarity, we

---

(1) We can relate these two layers to the Multics VM as follows: the LB layer corresponds to the Multics page fault handler, and the LBS layer would correspond to a subsystem that handles page faults on page tables.

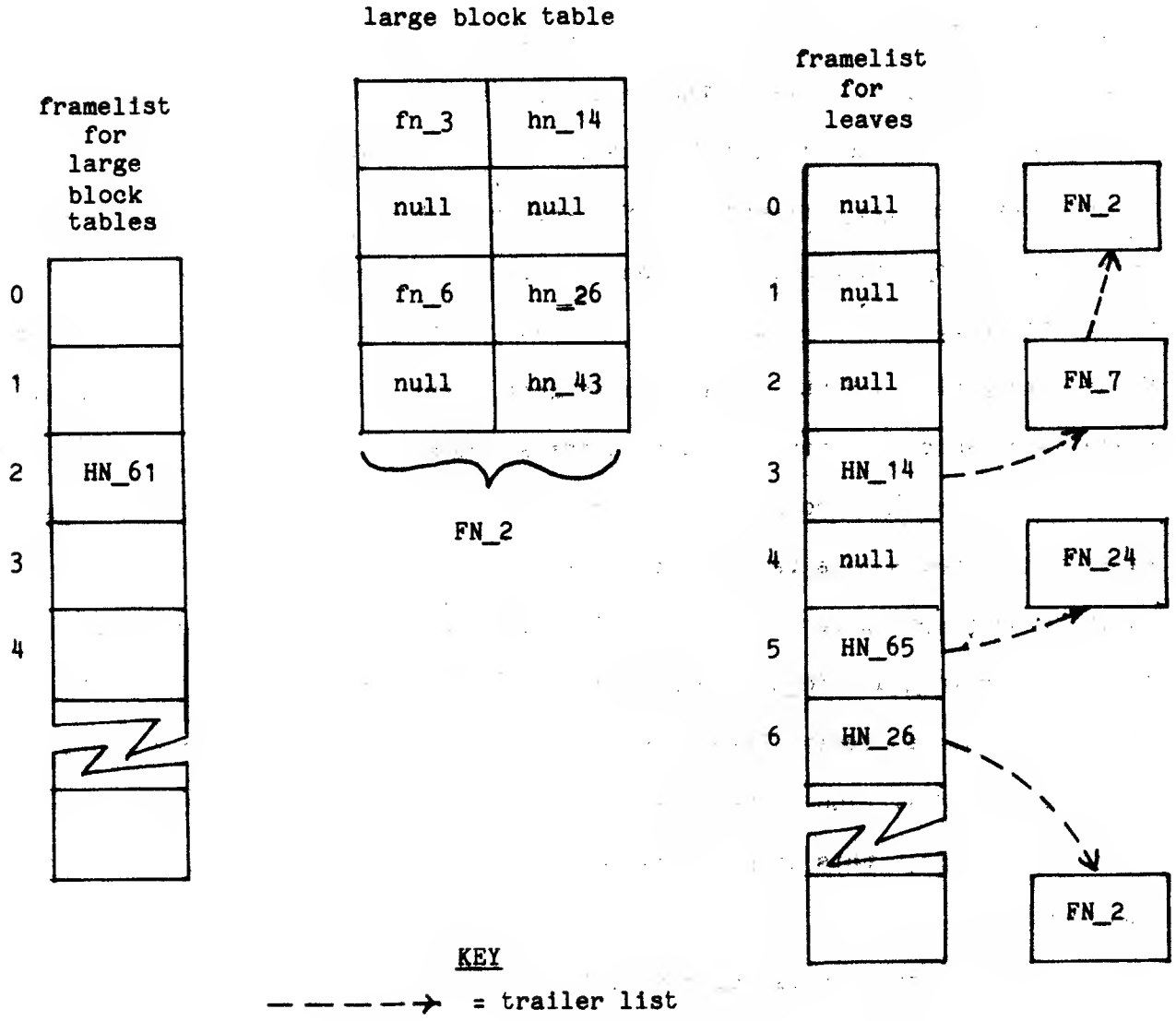


Figure 3-6

Interaction of Two Layers Over Frame Objects

use upper case "FN" to mean "frame name" in the case of frames that contain LB tables, and lower case "fn" to mean "frame name" for the leaf node frames. In the figure, we show two framelists -- one for each class of frame objects. Suppose that the LBS layer chose to free FN_2, depicted in Figure 3-6. The frame FN_2, however, contains the frame bindings for several pieces of a large block; e.g. the frame binding for the first piece is fn_3. When the frame freer of the LB layer selects fn_3 for freeing, it must be able to cause any reference to fn_3 that appears in an LB table to be deleted. It would be too costly to retrieve an LB table from secondary memory merely to set one of its frame name parts to NULL; therefore as long as an LB table has any non-null frame name parts, it should remain in primary memory.

We propose that the LBS layer have the option of freeing a frame, such as FN_2, that appears in a trailer list. It could do so by signalling the LB layer to undo appropriate bindings. In this example, the LB layer would replace both fn_3 and fn_6 by NULL, and remove FN_2 from the trailer lists of fn_3 and fn_6. Following these steps, the LBS layer could perform a store operation on FN_2. The correspondence between a frame, such as fn_3, and a home is not lost; this correspondence is still retained, by the LB layer, in the framelist.

The LBS layer supplies operational objects -- blocks -- that contain the representation of large block objects. If the LBS layer claims a frame, such as FN_2, that supports one of these blocks, and does so without informing the LB layer, the LB layer can fail to meet its specification. The LB layer is therefore strongly dependent on the LBS layer. However, the LBS layer

operates independently of the LB layer. It simply supplies information containers to the LB layer.

3.16 Further Aspects of Large Block Implementation

For completeness we include a brief description of the part of the LB layer that supports the growing and shrinking of large blocks. The role of this sublayer is to interpret a "raw" LB table in such a way that it produces a variable-length object with the properties that we specified earlier; e.g. that the initial value of any part of the LB with address less than the current length is zero. We include this section primarily to illustrate that, to provide a variable-length object, only a small amount of mechanism need be built on top of the common mechanism shared by the LBS and LB layers.

There is a sublayer of the LB layer that distinguishes among several types of "NULL" that may appear in the frame name part of an LB table. In particular, three kinds of "NULL" are

1. NULL(1): no corresponding home name;
2. NULL(2): there is a corresponding home name; and
3. NULL(3): beyond the current length.

We provide a scenario to illustrate how these interpreted values of NULL are used. Initially, an LB table would have the first K frame name parts containing NULL(1) and the remaining M-K containing NULL(3). (1) Suppose that a write operation occurs, directed towards a piece of the LB that falls within the scope of a NULL(1). In this case, a new home needs to be allocated. Note that if a secondary storage quota checking mechanism exists, it should be

---

(1) The granularity of the current length measure is only as fine as the frame size of the leaf node frames.

invoked at this time. Assuming sufficient quota, the LB layer does an allocate operation to get a home name, and then does an initiate operation, associating the home with the appropriate piece of the large block. At this point, the frame-claiming (and freeing) mechanisms are invoked to provide a frame for this piece of the large block. At some later time, if the frame is freed, the frame name part would contain NULL(2).

A read operation in the scope of a NULL(1) returns a value of zero, whereas a read in the scope of a NULL(2) generates a frame fault. A read or write operation in the scope of a NULL(3) generates an error condition.

The set_length operation merely moves the boundary between the NULL(3) entries and the other NULL entries. Shrinking the length of a large block may involve some terminate operations, since homes may have been associated with the section of the large block being truncated.

The LB layer can provide objects that are growable and shrinkable, and that are implemented using substantially the same data structures that support objects of non-varying length. The mechanism that supports the block abstraction is also the greatest common mechanism for the LBS and LB layers.

3.17 The Relation of the Large Block and Home Allocation Layers

Like the block layer, the large block layer simply manipulates the names of frame and home objects. However, part of the data base of the LB layer -- namely the set of LB tables -- must: 1) ultimately reside in secondary memory due to its size, and 2) grow and shrink dynamically since large blocks can be allocated and freed. Thus, unlike the block layer, the LB layer relies on the home allocation layer to manage the resources out of which some of its data bases are built. If the home allocator were to allocate the same home

twice, to two different large blocks, the specification of the LB layer would not be met.

The home allocation layer, however, does not depend on the LB layer. The home allocation layer executes in a block space, and embodies no knowledge of large block objects. As mentioned before, though, the home allocation layer does depend on the block layer.

Although there is no intrinsic reason why the home allocator should depend on the LB layer, there is a possibility of accidental dependency because both layers are consumers of the block layer. This is a specific case of the more general problem in which two layers depend on the block layer to provide an addressing environment. Either layer could initiate a block that properly belongs to the other layer and modify it -- thus introducing a two-way dependency. If two layers intentionally share information, and each trusts the other to "do the right thing" with the data, they are necessarily interdependent. In this case, it may be argued that they are really one layer. This form of dependency is obviously intrinsic. On the other hand, system designers may wish to provide mechanisms that prevent accidental interactions among layers. Consumers of a VM subsystem may rely on ACLs as a mechanism to prevent accidental interaction. However, in the lower layers of the VM itself, where ACLs are not available, another mechanism must be used. The mechanism we propose is the storage protection key mechanism, which exists in the IBM 370 series [IBM73]. VM layers such as the LB layer and the home allocation layer would have an associated key, and each home object would have an associated mask. The mask could be stored, for example, as a header word preceding the first data word in the home object. A layer could associate a

home with a block in its block space only if the key and mask satisfied some predefined relation.

3.18 One More Application of the Block and Home Allocation Layers

An objective of this chapter is to describe a particular implementation of a VM abstraction that can be characterized as a potentially large number of spaces containing a potentially large number of potentially large objects. At this point, it should be apparent that one more application of the block and home allocation layers should yield an implementation of this abstraction.

In this case, each LBS table that describes a set of LB tables should be implemented as a block object. Thus, we add one more level to the implementation tree, as shown in Figure 3-7. The incarnation of the block layer that manages blocks containing LBS tables is called the large block space space (LBSS) layer. That is, it manages a space of large block spaces.

As mentioned previously a small number of principals that are part of the supervisor may use a block space for an addressing environment. A second set of principals uses an LB space as an addressing environment. There is a one-to-one correspondence between each principal in this second set and an LBS table. There is thus a need to allocate and free homes that contain LBS tables corresponding to principals. The home allocation layer performs this function. Accordingly, the LBS layer depends on the home allocation layer.

The block layer mechanism described in this chapter serves as a greatest common mechanism for the LB, LBS, and LBSS layers. Since the structure of the LBSS layer is, by design, like that of the LBS and LB layers, previous observations regarding intra- and inter-layer dependencies apply to the LBSS

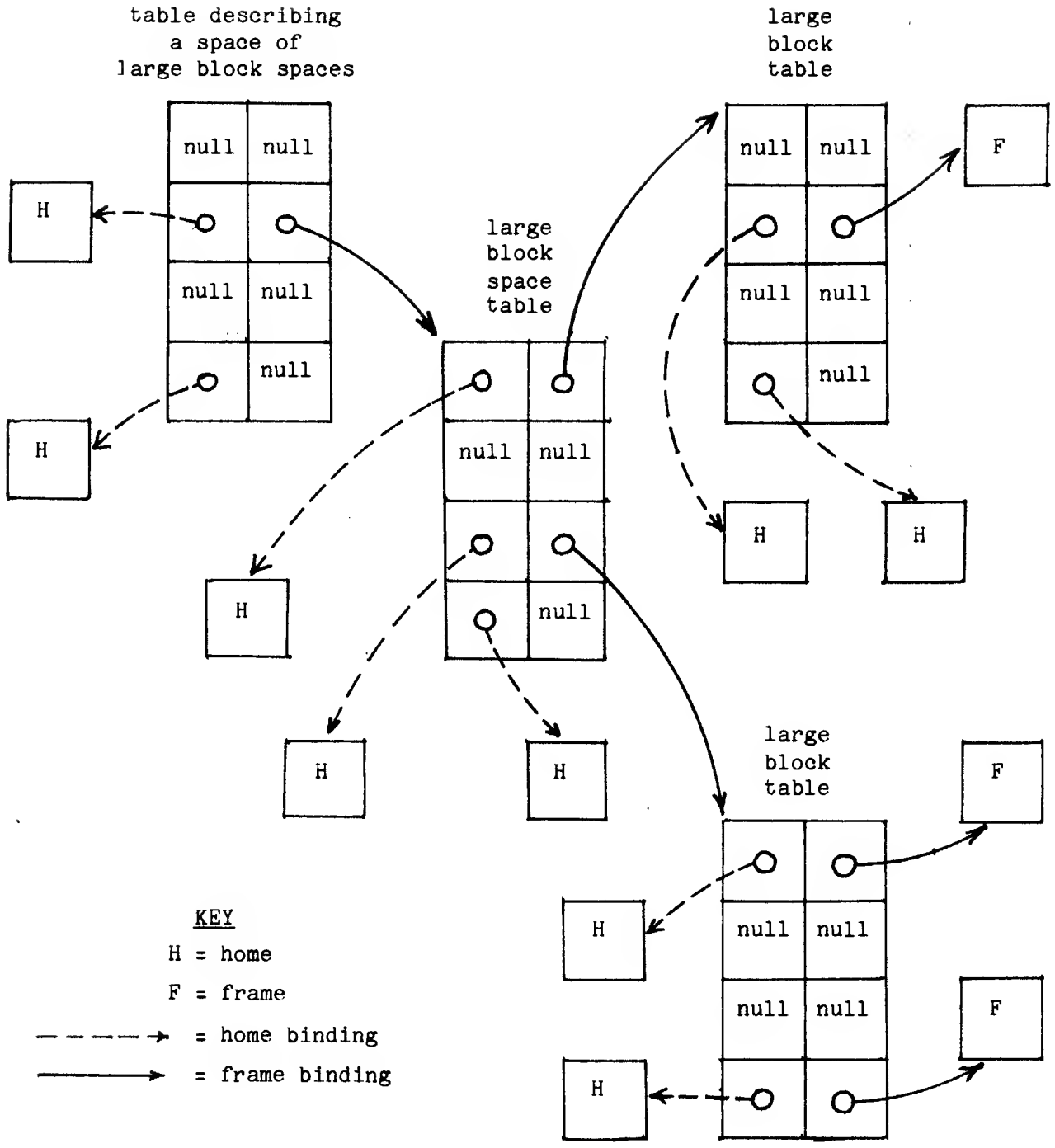


Figure 3-7

Data Structures Supporting a Space of Large Block Spaces

layer as well. The dependencies that relate each of the regions described in this chapter are shown in Figure 3-8.

3.19 Summary

In the lower layers of the case study VM, the primary function is the multiplexing of either real main memory or virtual memory objects. We have developed a model that characterizes memory multiplexing as the manipulation of bindings among a few simple object types. From the viewpoint of the model it is apparent that a type manager for any of these simple objects need not embody knowledge of the semantics of the other types. A feature of the multiplexing model is a high degree of modular independence.

Proceeding from the model, we show that a straightforward implementation preserves much of the desirable independence. This implementation can be supported by a hardware architecture similar to architectures of contemporary systems. Future architectures, which may incorporate hardware-assisted associative searching, should be able to support implementations that are truer to the model, i. e. there should be fewer implementation-induced dependencies.

The intermodule dependencies of the VM layers described in this chapter are either: 1) weak (i.e. timing) dependencies, 2) dependencies on an addressing environment, or 3) dependencies that occur because the specification of a layer embodies assumptions about objects that the layer references; for example, there is an assumption in the block layer specification that the data binding of a block object will not change unless a write operation occurs. The dependencies that do exist form a partial order among the modules of the case study VM subsystem.

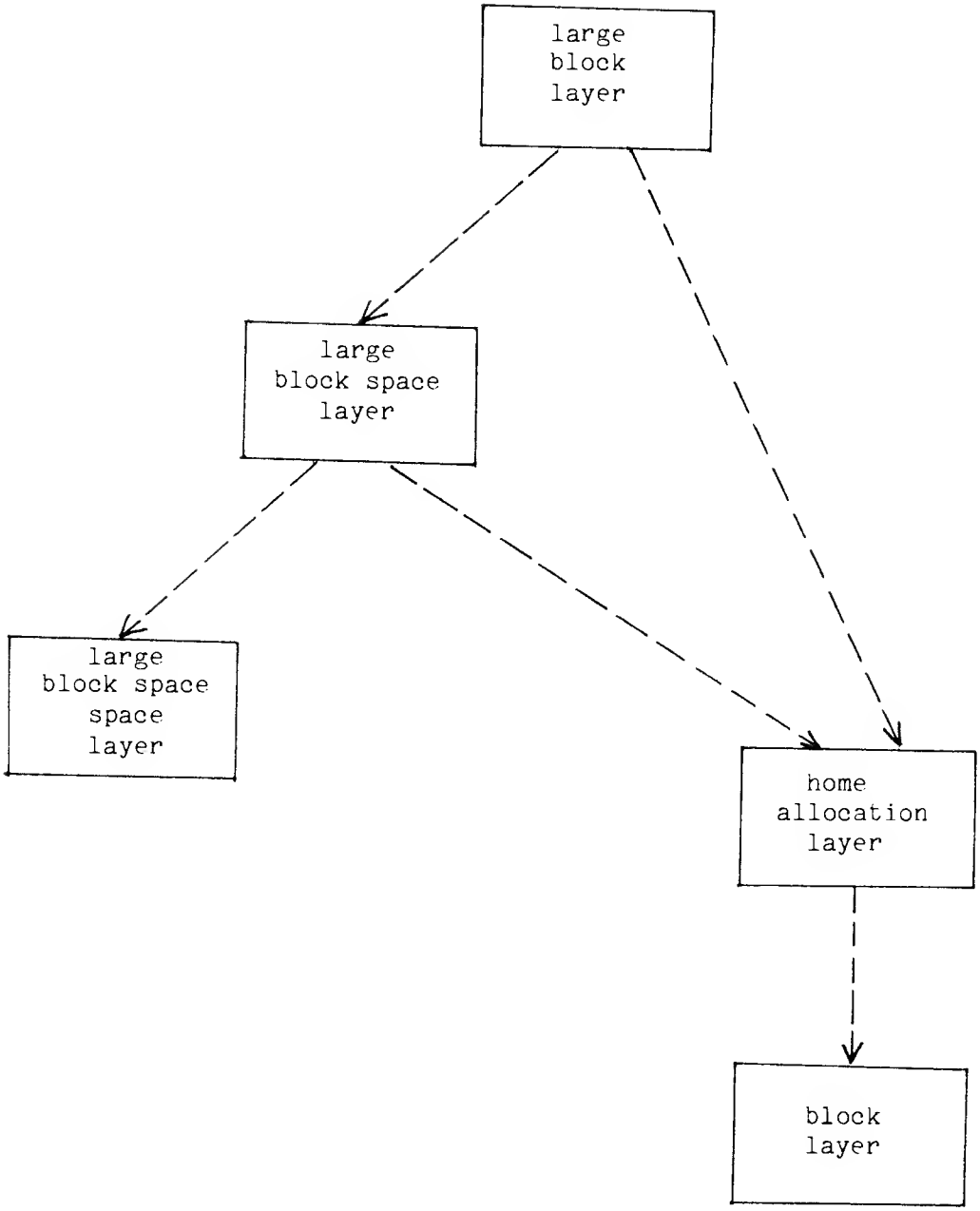


Figure 3-8

Dependencies Among the Lower VM Layers

Chapter IV

Treating Objects as Elements of a Property List

4.1 Introduction

In the previous chapter we described an object-oriented structure for the lower layers of the case study VM, in which relationships between objects took the form of LISP bindings. Even though bindings may exist between two different object types, the correct behavior of their respective managers can be independent.

In contrast, the higher layers of the VM resemble a hierarchy of type managers in which the correct operation of the manager of a type depends on the correct operation of the managers of its component objects. This chapter explores a method, based on the LISP concept of a property list, to minimize intermodule dependencies and achieve economy of mechanism in the higher VM layers.

4.2 Removing Unnecessary Dependencies in the Higher VM Layers

We observe that many of the operations in the higher layers of the VM are mapping operations; i.e. a significant function of each layer is merely to return the attributes of an object given its name. To minimize unnecessary inter-layer dependencies, it might appear that each layer would need to implement its own mapping function. However, we show that many layers can rely on a common layer to provide multiple mapping functions. This scheme certainly provides a greater economy of mechanism, and at the same time, the scheme can be implemented in such a way that the common layer does not depend

on the other layers. The common layer treats the range of values of each mapping function as elements of an uninterpreted property list, rendering it insensitive to anomalous behavior of the other layers. Of course, the other layers do depend on the common layer. In this sense this scheme is not as powerful as the binding scheme of chapter III, which has the potential of eliminating all dependencies between object managers. On the other hand, the property list scheme seems more generally applicable than the binding scheme. The goal of this chapter is to present a structuring method that is more appropriate for reducing inter-region dependencies in the higher VM layers.

4.3 Plan of the Chapter

In this chapter we justify the desirability of implementation-independent object names, and in particular the desirability of such names for segment objects. We describe a layer, called the map layer, that associates the implementation-independent names with the segment representations. The segment is represented by large block (LB) objects. Since the map layer manipulates some potentially large data bases, some form of underlying memory managemant function is needed. We show that the block and home allocation layers, described in chapter III, are sufficient as well as convenient.

The next section of the chapter describes how the map and LB layers together provide a viable substructure for any of several reasonable segment addressing mechanisms. Since addressing mechanisms like those described exist in current systems, this section should provide increased confidence in the viability of the VM layers described in preceding sections of the thesis.

This chapter then focuses on the implementation of access control lists, as an example of a class of nontrivial segment attributes. The ACL is

nontrivial since it is a potentially large attribute. We describe a layer called the ACL layer that supports the various ACL operations. As will be shown, some functions that are logically part of the ACL layer may nonetheless be provided by the map layer without causing the latter layer to depend on the former. These observations regarding ACLs can be generalized to other object attributes.

We next show that the VM layers that have been specified up to this point can provide effective support for a generalized type extension facility. This is because a common mechanism for supporting ETMs has already been provided for the support of segment and ACL objects. As mentioned in chapter II, a function that every ETM must perform is to map names to attributes. The map layer can provide this common function without becoming dependent on any ETM.

For completeness, we conclude this chapter with a brief discussion of 1) the representation of authority hierarchies in this VM structure, and 2) the implementation of directories as extended type objects. These two features can be provided without complicating the structure of the supporting VM layers.

4.4 Extending Large Blocks to Segments

The VM abstraction provided by the mechanisms in chapter III can be characterized as a potentially large number of spaces containing a potentially large number of potentially large objects. Further layers of the VM to be described in this chapter extend the above abstraction to that of a segment object. Before describing the additional layers, we review some characteristics of a segment object, as defined in chapter II, that still need to be provided.

First, the names of segment objects, unlike the names of LB objects, are implementation - independent. In particular, we are considering a segment name to be a unique identifier (UID), derived by reading a high-resolution clock. A segment UID may be bound to some representation object or objects -- in this case a home object containing an LB table. The advantages of such implementation - independent names are several. Since naming is distinct from implementation, the implementation can be changed "underneath" the names. Thus, a facility for backup of segment objects [Stern74, Benjamin76] can move the representation of the segment from one secondary storage device to another in a manner that is invisible to consumers. In addition, once a segment is deleted, the resources devoted to its implementation may be freed. It matters not if consumers retain the UID of the deleted segment, since implementation resources will never again be associated with that UID. The segment UID is an example of a non-recyclable name. A discussion of other advantages of using UIDs as segment names appears in Bratt [Bratt75] and Fabry [Fabry74]. (1)

Second, segment objects are expected to have a set of attributes. Examples of segment attributes are the name of the creating principal, and an access control list (ACL). The ACL is an example of an attribute that is potentially large. Although such an attribute requires more support mechanism than an attribute that is small and does not vary in size, we must be able to

---

(1) A less obvious advantage becomes apparent in the context of secure systems. If an objective of the system design is to minimize unauthorized information channels, then UIDs derived from clock readings are a good choice for object names. If the object name contained any information about the implementation or number of existent objects, then a user might be able to infer whether any objects were created between the times that he created two objects. Thus such a naming scheme would provide an unauthorized information channel (called a covert channel by Lampson [Lampson73]). The UIDs, on the other hand, provide no information to the creator other than the time of the creation.

support it, as it is part of our specification of a segment object. The underlying large block layer may indeed provide attributes, such as the current length value described in chapter III, that show through to the segment interface. However, the LB layer would not be expected to provide any potentially large attributes for LB objects, since the LB objects that it implements are precisely the most primitive objects that could conveniently support these attributes. Thus it is up to the higher VM layers, as consumers of the LB layer, to provide these attributes for segment objects.

There are other characteristics of a segment that appear in some implementations, such as automatic increase of the current length by a write operation, that can be built on top of the large block abstraction. We shall concentrate, however, on segment naming and segment attributes in this chapter since 1) it is our belief that these are two of the most distinctive characteristics of segment objects, and 2) we are able to describe an implementation of these characteristics that exhibits few interdependencies with other regions.

4.5 The Map Layer

The next VM layer to be described, the map layer, provides support for segment naming as well as for segment attributes. The map layer may be a consumer of abstractions supplied by layers that we have already described. However, the map layer is not a consumer of segment objects. If it were, there would be a circular dependency within the VM structure since the map layer and the type manager for segments would be mutually dependent. Such a circular dependency does occur, for example, in the Multics system since directory objects, which are built out of segments, implement the mapping from

segment names to segment representations. The case study VM, with its map layer, has no such dependency.

For purposes of explanation we begin by describing a mechanism that can support the segment naming function, and later extend that mechanism to support segment attributes. Thus the present concern is to describe the mechanism that maps each segment name, i.e. UID, into a corresponding home containing the appropriate large block table.

The map layer maintains a data base that associates segment UIDs with home objects. Following the terminology of Redell [Redell74], we refer to the data base as the Map. We have chosen here to use the capitalized form of the word, to distinguish this data base from any other data base that can be described by the generic term "map". The main requirement that the Map must satisfy is that any given lookup be rapid; although efficient insertion and deletion are desirable, they are of secondary importance. The Map must be able to provide the UID-to-home association for a very large number (say at least 10 million) of segments. Consequently it is an example of a large data base that must ultimately reside in secondary memory.

There are viable alternatives for the implementation of the Map. The Hydra system employs two hash tables. One hash table is in primary memory and the other is in secondary memory. The CAL system made use of a "master object table", residing in the extended core storage of a CDC 6400. Among other things, CAL capabilities contained indices into the master object table, providing for fast access to representation objects. In the case study VM, we suggest a B-tree [Knuth73] as the implementation of the Map. A B-tree is a balanced n-ary tree for which searching, insertion, and deletion operations have a guaranteed worst-case efficiency. It is a data structure that is

well-suited for external searching since each node can be implemented as one secondary storage record.

To maximize economy of mechanism, the lower VM layers already described should be employed to multiplex main memory among the B-tree nodes. Each B-tree node is accordingly implemented as a home object. The leaf nodes of the B-tree are simply homes that contain LB tables. The interior nodes are homes containing data with the format shown in Figure 4-1.

<home name> <UID> <home name> <UID> . . . <UID> <home name>

Figure 4-1

Contents of a B-tree node

Each home name in an interior node is the name of some other node. The UIDs surrounding a home name in Figure 4-1 are the lower and upper bounds on all UIDs reachable in the subtree with the given home name as the root node.

The B-tree structure supports a Map for a large number of objects at low cost. For example, suppose that the Map must accommodate 100 million segment objects. The home corresponding to a given UID can be located in 3 references to secondary storage if the B-tree is of order 100 (or more), and if the root node is already in primary memory. The average search time can be reduced if an associative memory containing (UID, home) pairs is provided.

The map layer can treat each of the B-tree nodes as blocks in a block space. For example, whenever it selects a home name from one of the interior

nodes during a tree search, it invokes the initiate operation to bind a block to the selected home. It then references the block directly to find the next home name in the sequence. The map layer can manage its block space so that nodes near the root of the B-tree tend to remain bound to blocks.

It is possible to regard the map layer as a type manager; it manages a collection of objects of type "name". The relevant operations are to make names known or unknown to the map layer, and to add, delete, or retrieve attributes of the known names.

4.6 Dependencies of the Map Layer

The block and home allocation layers described in chapter III are sufficient to provide the memory management support for the map layer. No special-purpose memory manager is needed.

The map layer strongly depends on the block layer since each node of a major data base of the map layer (i.e. the Map itself) is implemented as a block object. In addition, the map layer depends on the home allocation layer since it must allocate new homes to grow the map. The strong dependencies of the map layer are thus the same as those of the large block layer (see chapter III) even though the specifications of these layers are quite different.

Since the addressing environment of the map layer is a block space, the map layer and other layers that use this addressing environment could become interdependent. To provide some controls on the contents of a block space, storage protection keys, described in chapter III, can be used to prevent any layer from accessing programs and data private to other layers.

We emphasize that although the leaf nodes of the Map are data bases manipulated by the large block layer, the map layer does not depend on the

large block layer. The role of the map layer is merely to return the name of a home containing such a data base. The actual data base is never referenced by the map layer.

4.7 Alternative Addressing Modes for Segments

The purpose of this section is to illustrate the viability of the case study VM mechanisms. These mechanisms can support any of several segment addressing modes, including those of the Multics and Plessey 250 systems.

We have described two mechanisms that, when used together, allow a consumer to reference the contents of a segment object given its UID. The first mechanism is the map layer, which returns the home containing the LB table for a segment, given its UID. The second mechanism is the LB layer, which associates an LB object with such a home, and which provides an interface for referencing the contents of the LB. There are a number of ways that the facilities provided by these two layers may be combined to provide an interface for addressing segments. At one end of the spectrum of choices, all references to a segment object would be by its UID. At the other end of the spectrum, a local machine-oriented name could be associated with the UID, and thereafter all references to the segment object (from within that local context) would specify the local name. (1) We illustrate how the VM layers described so far can support several addressing modes chosen from this spectrum of possibilities.

The home containing the LB table for a segment is in essence an underlying state object; the large block that is used to contain the

---

(1) The justification for local names, as well as a discussion of alternative scopes for local names, appears in the work of Bratt [Bratt75].

information in a segment is an operational object. Together these objects are used to provide a segment object. Since a segment is named by UID, there must be a way to reference the supporting operational object given the UID. A mapping must be provided from segment name to supporting large block object. We define the segment layer to be the layer that performs this mapping, shown in Figure 4-2.

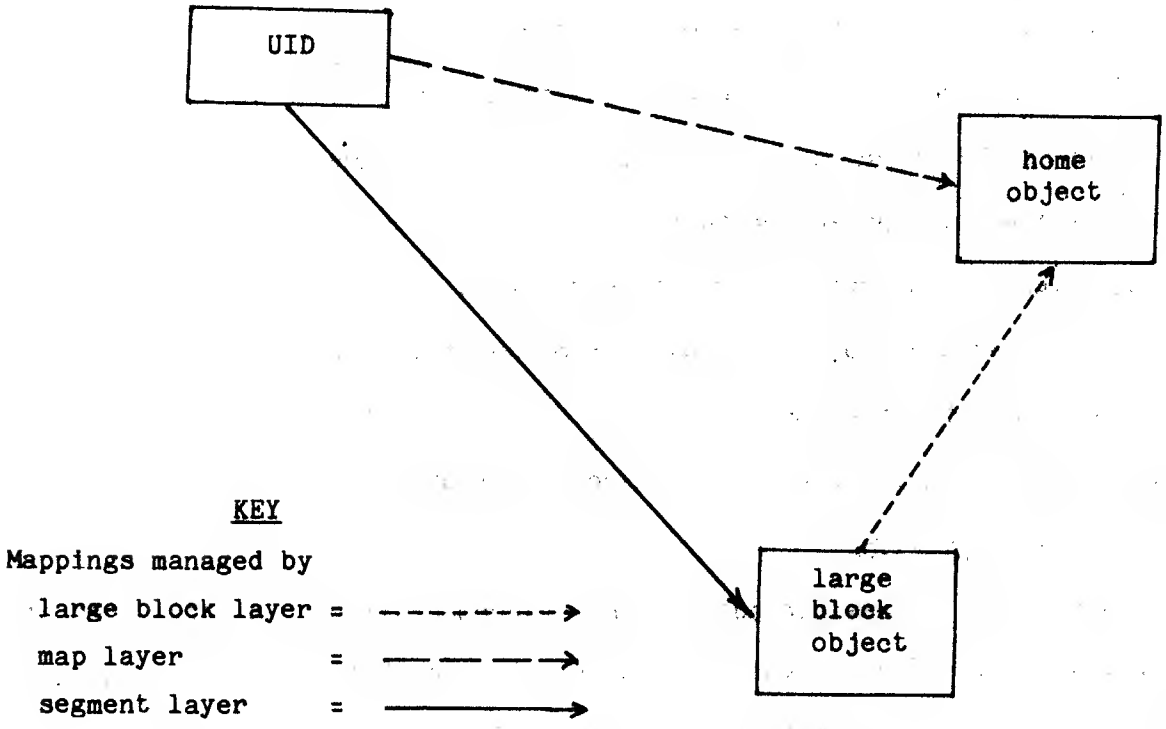


Figure 4-2

Mappings Managed by Three Different Layers

The segment layer is the type manager for segment objects. Given the name of a segment, it retrieves the components. It supports operations on segments (i.e. read and write) by associating underlying operational and state objects, and mapping segment operations into operations on the underlying operational object.

The implementation of the segment layer may rely on some optimizations to achieve efficient operation. For example, instead of mapping UIDs to large block names, the segment layer could map UIDs to frames containing large block tables. In order to support this more efficient mapping, the LB layer would need to include in its interface to the segment layer an operation that, given an LB name, would return the corresponding frame name. Such an interface would cause the LB layer to become dependent on the segment layer, since the segment layer would read and write frames directly and thus change the state of a large block. The problem is not very severe in this case, however, since: 1) the segment layer will hide frame names from all layers above it, and 2) the segment layer (which is expected to be implemented in hardware) is exceedingly simple and therefore the assertion that it hides frame names should be easy to verify.

As mentioned, a possible addressing environment is one in which segments are named by UID. This addressing environment, sometimes called a universal address space, has the advantage that segment names are context-independent. The name of a segment can be passed, without any translation, as a parameter or in a shared data base from one subsystem to another. The universal address space is the most flexible addressing scheme in the spectrum. As Bratt [Bratt75] points out, the reasons for choosing some other point along the spectrum are technological, not intrinsic.

There are no implemented systems, described in the research literature, that support a universal address space. Such systems, however, have been proposed. In a recent report, Radin and Schneider [Radin76] describe a machine interface that includes a universal address space. Redell [Redell74] has suggested an implementation of a universal address space that relies on hardware- (or firmware-) supported mappings from object UID to object representation. The suggested hardware support consists of a hash table supplemented by an associative memory. The representations of relatively active objects appear in the hash table, and the representations of the most active objects appear in the associative memory. In this case, the associative memory would associate on UIDs, returning a frame for an LB table. The hash table would also map UIDs into frames containing LB tables. Together, these two structures would implement the segment layer.

An alternative addressing environment that is quite similar to the universal address space makes use of register numbers for local machine-oriented names. The processor would provide a set of base registers that could be loaded with segment UIDs (and perhaps with offsets as well). Programs could then load segment UIDs into the registers and refer to segments by register number. The mapping from register numbers to UIDs is carried out explicitly by programs referencing segments. An advantage of this alternative over the previous one is that programs can use shorter segment identifiers. However, an identifier in this case is context-dependent so it must be translated if the object it designates is to be referenced in another context. Fortunately, the translation from UID to register number -- or vice versa -- is an inexpensive operation (i.e. a "load register" or "store register" instruction).

This scheme requires some form of hardware support, such as the associative memory and hash table of the previous section, to map segment UIDs into frame names. Alternatively, shadow registers associated with each of the base registers can be loaded with frame names of LB tables on demand. In either case, these forms of hardware support provide the function of the segment layer. This method of using base registers is similar to the approach adopted for the Plessey 250 system.

Finally, we mention the alternative of allowing large block names to be visible above the segment layer. In this case, programs would invoke an "initiate" primitive similar to that of the LB layer:

initiate (local_segment_name, UID),

in which "local segment name" is an input parameter, i.e. a segment index in a local segment space. Thereafter, all machine instructions would reference the segment by its local name. The shortcut employed in this case to make segment referencing efficient is to ensure that the underlying LB name is the same as the local segment name. A consequence of using this shortcut is that the role of the segment layer is diminished. It simply implements this high-level initiation primitive as follows:

1. it invokes the map layer to get a home name, H, given the parameter "UID", and
2. it invokes the LB layer to associate an LB name, namely "local_segment_name", with the home H.

The important observation here is that since the local segment name and the underlying large block name are arranged to be the same it does not matter, in the case of executing programs, whether this common name is bound to the segment UID. That is, after a segment object has been initiated, read and

write operations can be interpreted directly by the LB layer as read and write operations on an LB object. It is necessary to retrieve the UID only if a context-independent name for the segment is desired. This alternative offers an advantage over the previous one in that the set of local names would be larger, since the number of large blocks in an LB space is assumed to be larger than the number of processor - supported base registers for segment UIDs. On the other hand, the cost of determining the UID that is bound to a local name is greater in this case than in the preceding case, since this operation invokes VM layers that would probably be implemented in software.

In order to relieve segment consumers of the necessity of managing local segment names, a simple layer can be built on top of (or in) the segment layer. This simple layer would implement some policy of assigning local segment names to UIDs, and would represent this assignment in a table that corresponds to each large block space table. Such a table would be similar in function to the Known Segment Table in the Multics system [Bensoussan72, Bratt75].

It has been the purpose of this section to describe how the map and large block layers, described previously, can be used by higher layers to provide any one of several VM addressing environments that are found in current and planned general-purpose systems. This description is motivated by a secondary goal of this thesis: to provide some justification that the oase study VM is a viable one.

4.8 The Access Control List Layer

Our definition of a segment object includes the presence of an access control list (ACL) attribute. An ACL has a property in common with a segment:

it is potentially large. This suggests that the mechanism for providing this property can be common to the implementation of both ACLs and segments. We present a design for the support of ACLs that maintains economy of mechanism on the one hand, and preserves strict layering modularity on the other. The design for the support of ACLs is general enough that it can support any attribute that is potentially large.

Methods for implementing small segment attributes are not appropriate for ACLs. In the case of the small attribute, the value could be stored in the LB table or in an expanded Map entry for the segment. For example, the current length attribute described in chapter III is represented in the LB table. However, since Map entries as well as LB tables are data bases that are small (i.e. they are implemented using block objects), they cannot contain the representation of an attribute like an ACL. Even though an ACL could be small, it may be as large as any segment object. Hence, the representation for an ACL cannot, in general, be stored either in a Map entry or in an LB table. Of course, the Map entry or LB table could contain some kind of pointer to the representation of a large attribute.

It is more difficult to implement a potentially large object, like an ACL or segment, than it is to implement one that is either large but constant size or variable length but small. Large representation objects will suffice for objects in the first class, and small representation objects will suffice for objects in the second. If an object may be either small or large, the supporting mechanism must be able to concatenate small objects, as necessary, to effect an implementation.

The subsystem that implements ACLs, which we shall call the ACL layer or ACL region, uses large block objects as the representation objects. Although

a segment object is a variable-length container that could represent an ACL, segments have ACL attributes of their own, so such an implementation would introduce a circular dependency between the segment and ACL regions. Even if this circular dependency were tolerated, some fixed point must be established. The large block object suffices as the fixed point since it is a potentially large container that has no ACL attribute. As a consequence of choosing the LB object to serve as the representation for ACLs, no circular dependency is introduced. This method for decomposing potential circular dependencies is called sandwiching by Parnas [Parnas76].

Access control list objects that are small can waste space in the supporting LB tables and home objects. However since the ACL layer allocates large blocks, as necessary, to represent ACL objects, it can represent many ACLs in the same large block. This strategy would eliminate the wasted resources caused by breakage, but it would require that there be a dynamic storage allocation facility within the ACL layer.

This method for implementing ACLs treats each ACL as a distinct object. Such a treatment is quite natural: any attribute of a given object may, at some lower level of abstraction, be considered a distinct object. The operations that can be performed on an ACL are search, display, and update. The search operation is invoked to determine whether a given principal can perform a particular operation on the associated object. The display operation lists ACL entries, and the update operation changes them.

On each segment reference, a principal must invoke the ACL layer to search the corresponding ACL. To provide for efficient reference to the ACL, part of the ACL layer is implemented in hardware. Corresponding to each principal's large block space table is a parallel table that comprises access

rights fields. Each access rights field contains a bit-encoding of the principal's access rights to the corresponding segment object. Every machine instruction reference to a segment is checked against the appropriate field of encoded access rights. (1) The access rights field is initialized the first time a principal references a segment. On the first reference, the hardware part of the ACL layer causes a processor fault, and the following steps are performed.

1. The ACL object corresponding to the segment object that was faulted upon is located in a system-wide table.
2. The ACL layer initiates the large block representing the ACL, if necessary, and searches it.
3. If access is not allowed, the ACL layer signals an access violation.
4. Otherwise, it sets the access rights field to contain the proper encoding of rights for the referencing principal.

4.9 Eliminating Potential Dependencies by Using Property Lists

In this section we consider two mappings, described in steps 1 and 2 of the preceding section, and illustrate how the map layer can be used as a greatest common mechanism to support both of them. The two mappings are

1. the mapping that, given a segment object name, returns the associated ACL object name, and
2. the mapping from ACL objects to their representation (LB) objects.

---

(1) The Multics system employs an encached form of access control list entry such as this. In the Multics implementation, the encached access rights and the encached addressing information are included in a single table called a descriptor segment.

Providing the second of these two mappings is a function of the ACL layer, the type manager for ACL objects. It is the standard mapping from object name to object representation. Providing the first mapping should really be a function of a layer that manages "segment-with-ACL" objects. This layer would provide a mapping from each such object into its two components: a "segment-without-ACL" object and an ACL object. All of these mappings are shown in Figure 4-3. Providing distinct implementations of either of these mapping functions appears to be a diseconomy of mechanism, since there already exists a similar mechanism in the map layer that relates object names to object representations. Other layers could use this mechanism in the map layer. Of course, neither the ACL layer nor any other layer should be able to manipulate the Map directly in order to provide these mapping functions, as this would violate layering.

We solve this problem by appealing to a technique that preserves strict layering on the one hand, while maintaining economy of mechanism on the other. The technique is derived from the LISP notion of a property list -- i.e. a set of uninterpreted attributes associated with an object. There is no violation of layering if a lower layer maintains data, in a property list, for a higher layer. The higher layer can attach a particular interpretation to the data. The lower layer can provide a mapping of the form

property name ----------> property value

for the higher layer. The higher layer has precisely two ways to interface to the lower layer, which are

1. fetch_property (object_name, property_name, value), and
2. store_property (object_name, property_name, value).

The only causes for error conditions are object names or property names

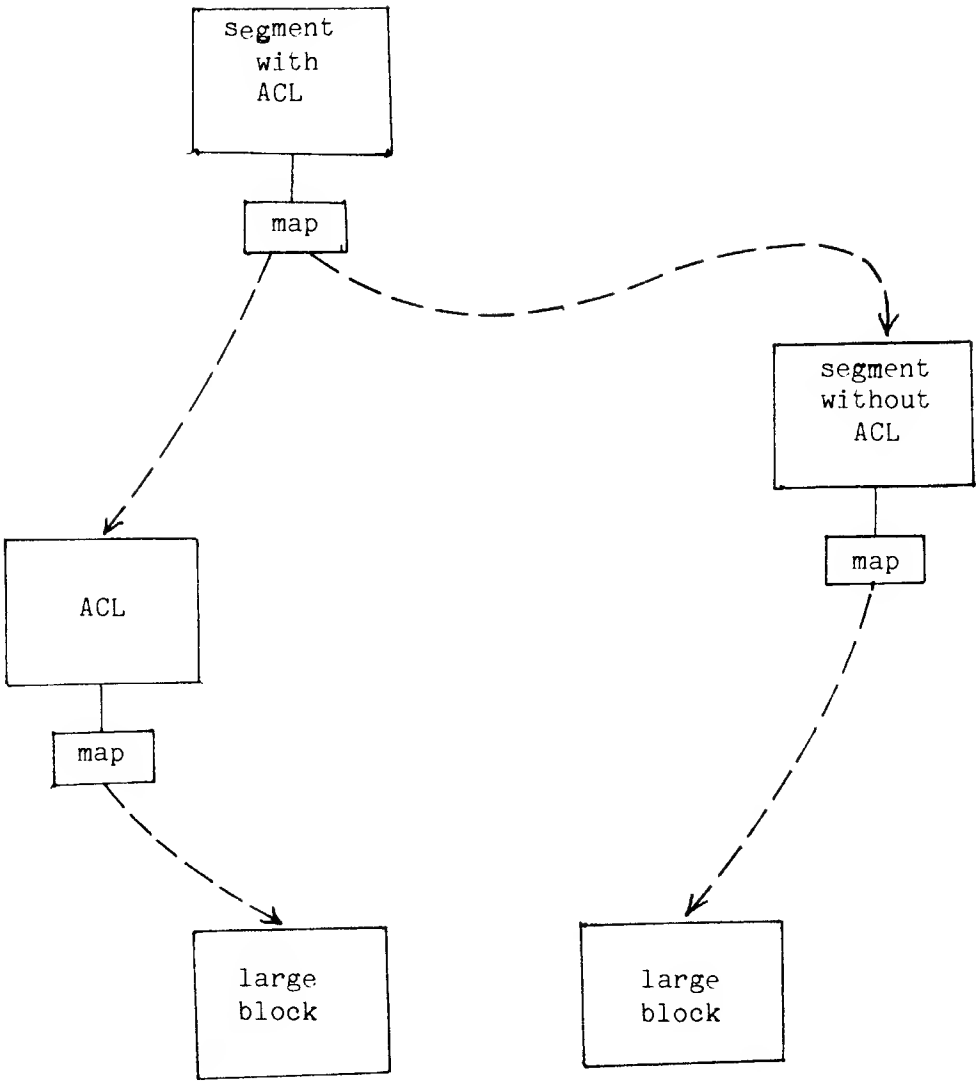


Figure 4-3

Possible Representation Objects for Segments

unknown to the lower layer. The correct operation of the lower layer does not depend on the correct operation, or even the existence, of the higher layer.

The map layer is in fact a layer that performs a

property name ----------> property value

mapping for a higher layer, namely for the segment layer. In this case, the segment layer invokes the map layer to obtain the "representation" property of a segment, by performing the operation

fetch_property (segment_UID, representation, value).

The value of "representation" returned by the map layer is the home name of the underlying LB table for the segment. Of course, the map layer places no interpretation on this value; rather it considers it to be a property of the object named "segment_UID".

Since, as mentioned previously, the segment layer is the type manager for segment objects, we see that the map layer actually implements the mapping from object name to object representation for the segment layer. In our characterization of type managers in chapter II, we indicated that every type manager is responsible, given the name of one of its objects, for locating the corresponding representation objects. The segment layer thus relies on the map layer as a utility subsystem, for performing this mapping.

The map layer can also be used by other type managers to locate representations, given object names. In particular, those mapping functions depicted in Figure 4-3 could be carried out by the map layer. However, not all those mappings need be realized, since there is no need to reference either "segment-without-ACL" objects or ACL objects directly. These objects are of interest only insofar as they are attributes of "segment-with-ACL"

objects. Thus, a superfluous set of names and mappings can be eliminated by treating both the LB representing a "segment-without-ACL" and the LB representing an ACL as two representation objects of a "segment-with-ACL". We can then rename the "segment-with-ACL" object, simply calling it a segment, with the understanding that all segments have ACL attributes. The number of mappings is now only two, as shown in Figure 4-4. We can then rely on the map

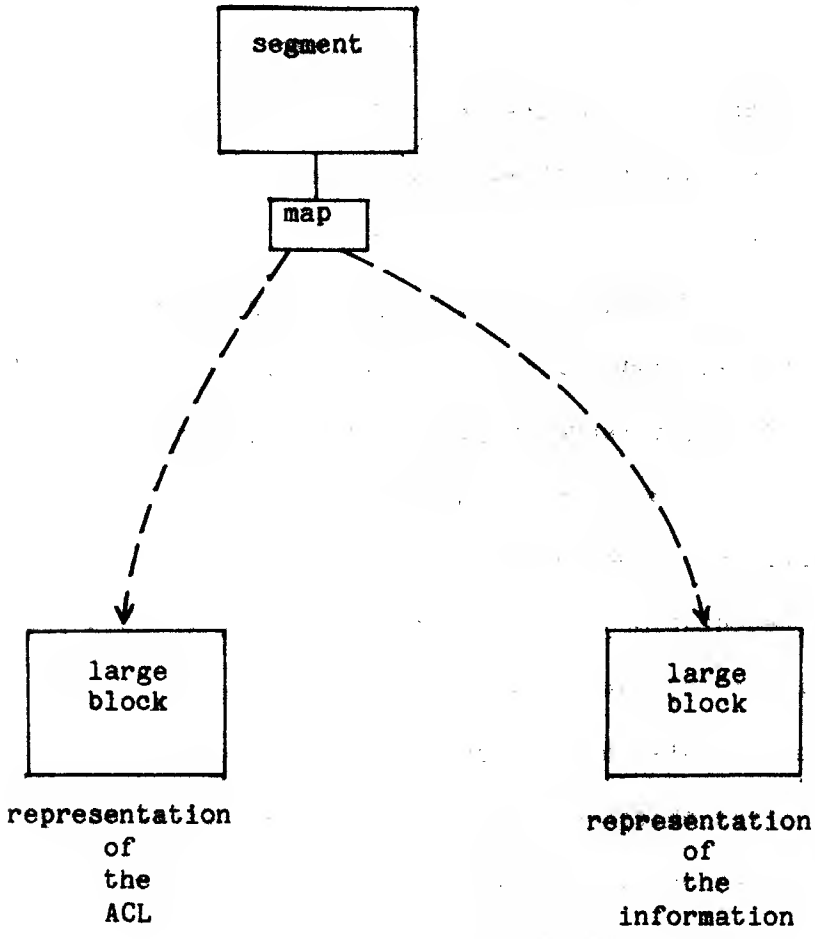


Figure 4-4

Actual Representation Objects for Segments

layer to implement the mapping to two representation objects of a segment: the representation of the information part of the segment, and the representation of the ACL attribute of the segment. Each Map entry for a segment object is thus expanded to contain the home names of two large block tables. The ACL layer can then obtain the representation object for the ACL of a segment by invoking the "fetch_property" operation described previously.

From the standpoint of the map layer, each segment object name has two properties: the ACL property and the representation property. The value of the ACL property, interpreted by the ACL layer, is the name of the home that contains the LB table describing the ACL. The value of the representation property, interpreted by the segment layer, is the name of the home that contains the LB table describing the information of the segment.

Some operations on a segment object, such as read and write, are implemented by the segment layer. Other operations, such as the operation to search an ACL of a segment, are implemented by the ACL layer. Operations on segment objects involve the segment and ACL layers as follows.

1. The segment layer is invoked in order to determine what bit-encoding of access rights corresponds to the operation (e.g. read) that is being requested.
2. The segment layer invokes the "search" operation of the ACL layer, passing the bit-encoding, the consumer name, and the segment UID as arguments.
3. If the ACL layer indicates that the search was successful, the segment layer performs the requested operation.

We have, in this design, treated the ACL information for a segment as a property in a property list maintained by the map layer. This design technique preserves the layered structure of the VM, since the ACL object -- a

good candidate for objecthood -- is indeed managed by its own type manager. At the same time, a duplicate mechanism for obtaining the names of representation objects has been avoided.

Even though ACL objects do not have distinct implementation - independent names, the programs of the ACL layer can be written under the assumption that they do. The UID of the segment can be regarded by ACL layer programs as the unique identifier of the ACL object instead. The ACL layer invokes the map layer operation

fetch_property (segment_UID, ACL, value),

causing the map layer to return the value of the ACL property. This naming technique could be applied to other attributes of segments which, like ACLs, require distinct implementation objects.

However, there are limitations to this approach. First, this approach is applicable only to objects that are used solely as attributes of other objects. For example, if an ACL were a free-standing object that could be referenced by arbitrary consumers, then it would need an implementation-independent UID for a name. Second, this approach is biased towards efficiency rather than towards generality. It is appropriate for supporting only a fixed number of attributes since, for each attribute, there must be a field in the Map entry of an object. To avoid inefficient Map searches, the format of a Map entry should be the same for all segments. Hence the size of a Map entry cannot vary, and neither can the number of attributes. In contrast, LISP systems are biased towards generality since they support large property lists for objects. Locating an arbitrary property typically involves searching a list structure, which would not be as efficient as referencing a Map entry.

4.10 A Layer to Support Dynamic Type Extension

This section illustrates how a layer for supporting dynamic type extension can be built on top of the layers already described. The type extension facility represents the last functional component of this case study VM subsystem. We intend to show that techniques for achieving modular structure and economy of mechanism at the same time, such as those employed to support the ACL layer, can also be used to support dynamic type extension.

As described in chapter II, each operation on an ETO causes 1) type information, 2) ACL information, and 3) component objects to be referenced by the ETM. The ETM may rely on some other subsystem to fetch this information. Nonetheless, each of these three kinds of information are obtained on every ETO operation. We specify a distinct layer, called the extended type manager (ETM) layer, that supports these common operations. The ETM layer is the first layer we have described that places any interpretation on type information.

The ETM layer of this thesis is quite similar to the Extended Object Manager layer of the SRI system. The SRI Extended Object Manager is responsible for storing and retrieving the component objects (called implementation capabilities) for each extended type object. The ETM layer of this thesis also stores and retrieves component objects. In addition, it stores and retrieves both ACL and type information. Since one of our goals is to describe a VM in which all types are protected by ACLs, mapping from object name to ACL information is indeed a mechanism common to all ETMs. (1) In order to reduce errors, the ETM layer also enforces two policies: 1) a type

---

(1) The preceding section described how the map layer could support such a mapping, but only for segment objects.

manager may search only those ACLs of its own objects; and 2) a type manager may request the component names of its own objects only.

The ETM layer can be considered to be the type manager for a data base object that contains type, ACL, and components information. Higher layers call upon the ETM layer in order to enter or to retrieve this information.

In the preceding section, we showed how the map layer can be extended to provide ACL information for segments. In following sections we show that the map layer can be extended further to associate object UIDs with the above three kinds of information, for any object type. The map layer can thus serve as a common mechanism for the segment, ACL, and extended type manager layers.

4.11 The Extended Type Manager Layer Interface

The three important interfaces to the ETM layer are listed below. The first interface could be invoked by any principal; however only ETMs will make use of the latter two interfaces.

The first interface, the mapping from object UID to type information, can be specified as follows:

get_type (object_UID, type),

in which "type" is an output argument. By invoking this interface, any principal may determine the type of any object. (1)

---

(1) As mentioned previously, attributes of objects, and even the existence of objects, may be viewed as covert information channels. In applications where this is important, type information should not be available to an arbitrary principal. This policy could be enforced by non-discretionary controls. In this case if a principal unauthorized to know about the existence of an object managed to guess the right UID and pass it as a parameter to the above interface, it would receive an error message of the form: "Either the specified object does not exist, or you are not allowed to know if it exists."

The second common interface returns ACL information. It is specified as

search_ACL (object_UID, supplier_UID, consumer_UID,
bit_encoding, boolean),

in which a "true" value of the boolean output parameter would indicate a successful ACL search. Since the type of an object is implemented as the UID of the managing subsystem, the ETM layer can check to make sure that the (unforgeable) value of "supplier_UID" equals the type of the object named "object_UID". This check ensures that only the type manager for an object will be able to search the corresponding ACL. If the UID of the supplier is correct, the ETM layer can invoke the "search_ACL" entry of the ACL layer, just as the segment layer does. The ACL layer will perform the search operation for any principal. Thus the check performed by the ETM layer merely separates type managers, according to the principle of least privilege.

The third common interface obtains component objects. It has the form

get_components (object_UID, supplier_UID, components),

in which "components" is an output parameter. Since the component objects of every ETO are either segments or other ETOs, the value of "components" is a set of UIDs. As before, the ETM layer checks that this mapping function is invoked by the correct type manager. This check is provided only for self-protection purposes, so that an ETM will not erroneously request the components of an object of another type. A malicious ETM might guess the components of any ETO; however those components are protected from unauthorized access by ACLs.

4.12 Support of the Extended Type Manager Layer by the Map Layer

Some mechanism within or below the ETM layer must support the mappings from object name to type, ACL, and representation information. The ACL information for ETOs is supplied by the ACL layer which, in turn, obtains it from the map layer. Type and component information for ETOs is supplied by the map layer.

The map layer can treat the ETOs and associated attributes as a set of objects with property lists. The Map itself will thus contain entries not only for segment objects, but for objects of every type. Each entry in the Map must then contain type information, in addition to the ACL and representation information previously specified. For any object, the map layer can return type, ACL, and representation properties. The revised format of a Map entry is shown in Figure 4-5.

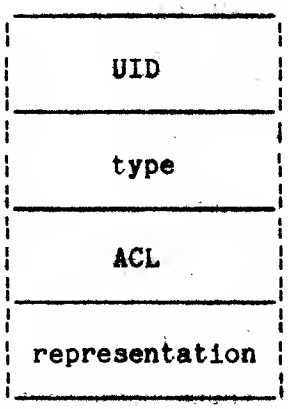


Figure 4-5

Revised Format of a Map Entry

For all objects, the "UID" field contains a system-wide unique identifier. For segment objects, the value of the "type" field is some flag indicating that the object is a segment, and the value of the "representation" field is the home name of the home containing the underlying LB table. For extended type objects, the "type" field contains the UID of the principal that is the type manager of the object, and the "representation" field contains the UID of some other object that serves as the representation object. (The representation object may, in turn, contain a list of UIDs of additional representation objects.)

The contents of the "ACL" field may take two forms. If the object is not a component of some other object, then the ACL field consists of the home name of the home containing the LB table for the ACL object. Otherwise, if the object is not a component of some other object, then the ACL field contains a degenerate ACL. The degenerate ACL, consisting of a type manager UID and mode bits, is sufficient in this case, as was pointed out in chapter II. In contrast to an ordinary ACL, it can be searched faster and requires no underlying potentially large representation object.

To tie these ideas together we show, in Figure 4-6, typical map entries for an ETO and a component object. Let a message queue object, "MQ_26", be managed by the type manager "MQ_mgr". The only component object representing MQ_26 is the segment "SEG_14". Located in the home object "home_43" is the LB table for the LB that contains the (non-degenerate) ACL for MQ_26. The ACL for SEG_14 is a degenerate ACL containing the sole principal identifier "MQ_mgr". The LB object representing "SEG_14" is described by the LB table located in "home_31". The Map entries for the two objects named "MQ_26" and "SEG_14" are shown in Figure 4-6.

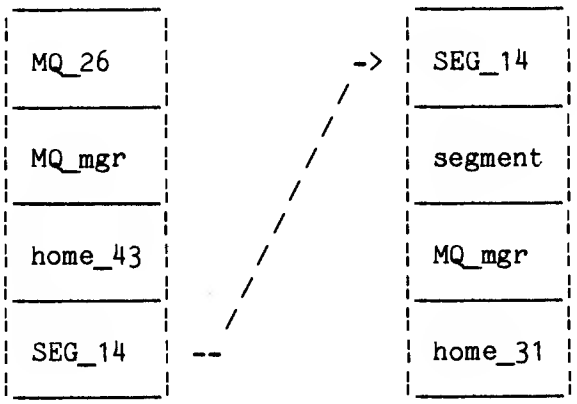


Figure 4-6

Map Entries for an Extended Type and a Component Object

4.13 Dependencies of the Extended Type Manager Layer

The ETM layer depends on the ACL and map layers to support its mapping functions. It depends also on the segment layer (or possibly the large block layer) to support its addressing environment.

A role of the ETM layer is to subdivide the set of all objects into types. Thus, it must associate types with objects. Based on type information, it makes ACL and component information available only to the appropriate type manager. If the underlying ACL and map layers incorrectly map an object name into an object attribute, the ETM layer cannot meet its specifications. It thus depends strongly on the ACL and map layers.

The ETM layer may depend on either the LB or segment layers to support its addressing environment. Since the ETM layer is not necessary for the

implementation of segment objects, it could execute in a segmented addressing environment. In this case it would be dependent on the segment layer. (1) An alternative implementation is that the ETM layer execute in an LB space. This suggestion is based upon the observation that since much of the functionality of the ETM layer is already provided by the map layer, the ETM layer could be implemented as an upper sublayer within the map layer and thereby execute in the same LB space. Given this alternative, the ETM layer would depend on the LB layer.

4.14 Representation of an Authority Hierarchy

The purpose of this section is to suggest how a simple authority hierarchy, as specified in chapter II, might be implemented in a way that does not complicate the VM structure we have described so far. In chapter II we indicated that a special class of protected subsystems, called offices, can implement administrative control over access control lists. We can consider each office to be the consumer of a set of ACL objects.

The kinds of operations that an office performs on an ACL are to display and update the local authority structure represented in the ACL. The principal identifier (i.e. a UID) of the controlling office of an ACL is contained in the ACL. A suggested ACL format is shown in Figure 4-7. The object-specific rights are in region 1 of the ACL. For example, in a segment ACL, rights for read, write, and execute would appear in this region. The ACL-specific rights, display and update, appear in region 2. One interpretation of these rights is that a principal with display rights can

---

(1) Certainly, layers above the ETM layer must execute in a segmented environment since they may be provided by arbitrary users.

display the contents of all three regions, whereas a principal with update rights can change entries only in region 1. A principal attempting to perform some operation on the associated object will succeed only if it is named in region 1. Only the principal appearing in region 3, namely the controlling office, can update entries in region 2. The contents of region 3 cannot be changed.

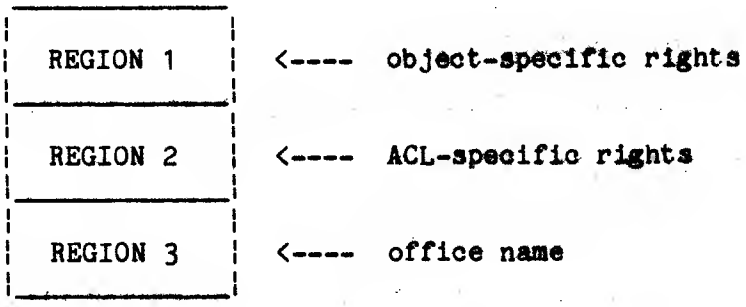


Figure 4-7

Contents of an Access Control List

In this implementation of an authority hierarchy, the ACL layer is the type manager for all ACL objects. The implementation details of ACLs are the sole responsibility of the ACL layer. However, the ACL layer will carry out the policy embedded in the controlling office of an ACL. The office, in some unspecified way, can determine that a display or update operation should take place, and invoke the ACL layer to effect it.

4.15 Directories as Extended Objects

We have shown how access control lists can be incorporated, as distinct objects, into a VM structure that is characterized by 1) a small number of intermodule dependencies, as well as 2) economy of mechanism. In this section we sketch how directories may be implemented as extended type objects, while still preserving these two characteristics of the VM structure.

As Redell [Redell74] points out, directories are typically implemented as ETOs in capability-based systems. However, in the ACL-based Multics system, directories are base level objects that contain physical descriptors, as well as ACLs, for other objects. References to Multics objects, mediated by ACLs, necessarily involve the directory layer. In addition, references to directory objects themselves are mediated by ACLs. The Multics directory and ACL mechanisms are mutually dependent. In the case study VM, we break this dependency by implementing directories as ETOs, with ACL objects implemented in a lower layer. The ETM for directory objects, the directory layer, depends (indirectly through the ETM layer) on the ACL layer, but not vice-versa.

As indicated in chapter II, directory objects associate alphanumeric names with object UIDs. Like all ETOs, directories have ACLs. The operations that appear in directory ACLs include "append", "display", "search", and "delete". Directory objects are implemented in terms of segment objects.

Since directories are implemented in terms of segments, the directory layer depends on the segment layer. Another reason for this dependence is that the directory layer -- a true extended type manager -- must execute in a segmented addressing environment.

Of greater interest are possible dependencies of lower VM layers on the directory layer. The correct operation of each individual layer up to and

including the ETM layer is independent of the correct operation of the directory layer, since no lower layer invokes the directory layer. For example, neither the segment layer nor the ACL layer depends on the directory layer.

The correct operation of the directory layer is nonetheless critical for carrying out intended naming and protection policies. The directory layer is responsible for mapping user-readable names into UIDs. Consequently incorrect operation of this layer may cause a consumer to read the wrong segment, or to update the ACL on the wrong message queue. If the directory layer associates names incorrectly, the lower layers that deal with objects named by UID will still function correctly; however the intended request of some higher layer will not have been carried out. It should be emphasized that there is no difference between ACL-based systems and capability-based systems in this regard: directories in capability-based systems, such as the SRI and Hydra systems, are used to associate names with capabilities and malfunction of this association may violate the naming intentions of a user.

4.16 Summary

In contrast to the VM layers considered in the previous chapter, the higher VM layers considered here carry out mapping functions from object names to object attributes. The challenge in this chapter is to describe a modular structure that both exhibits few intermodule dependencies and also minimizes costly duplication of the mapping mechanism. To eliminate duplicate mapping mechanisms, a greatest common mechanism -- the map layer -- is included in this VM subsystem.

The intermodule dependencies in the higher VM layers form a directed acyclic graph, as shown in Figure 4-8. To achieve this structure, we have

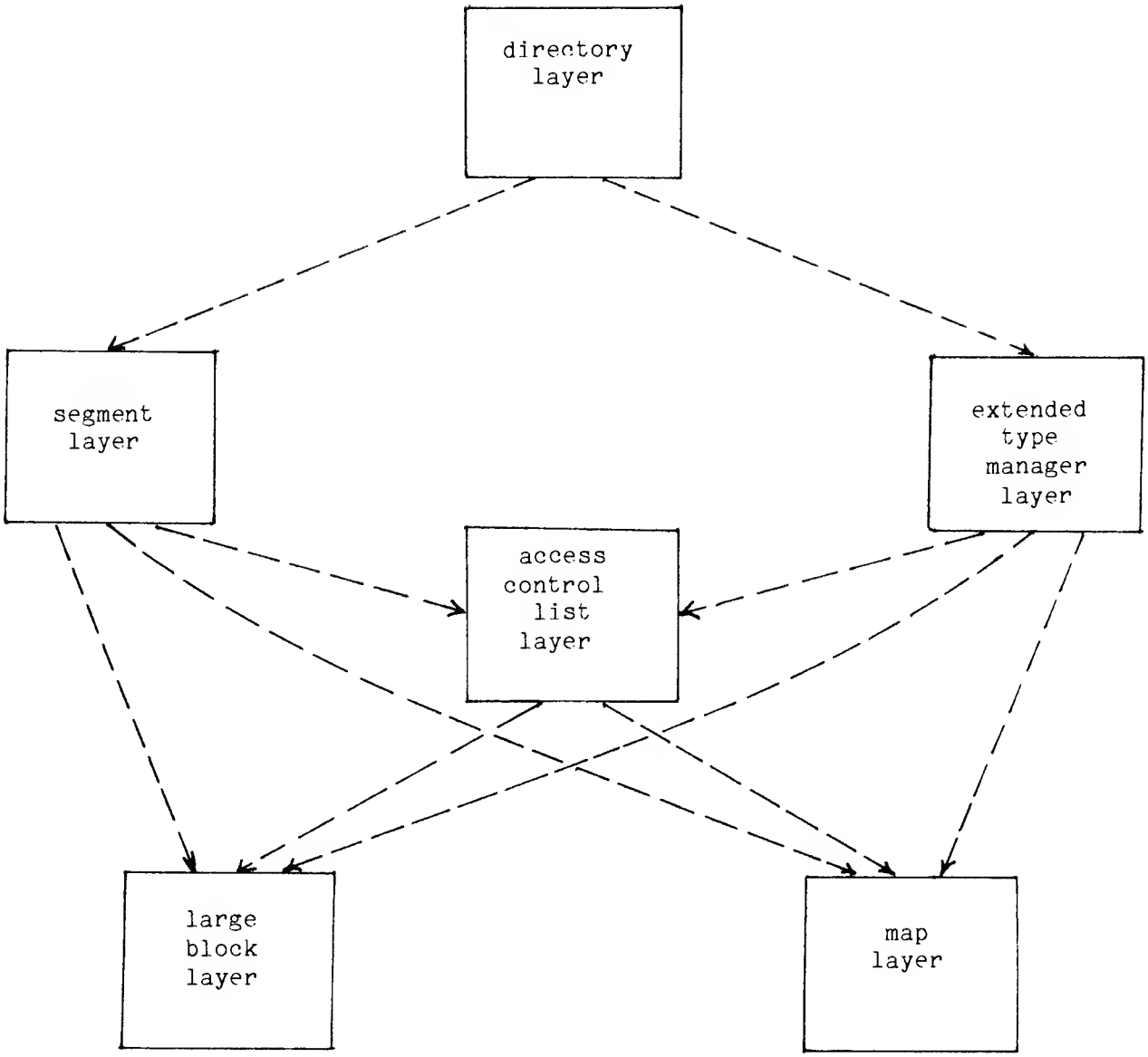


Figure 4-8

Dependencies of the Higher VM Layers

relied on the sandwiching technique, and, more importantly, on the LISP notion of a property list. The applications of the sandwiching technique described here are effective, but seem to be ad hoc. In contrast, the use of the property list notion seems to be a more generally applicable technique for eliminating potential circular dependencies.

Implementing ACLs as large block objects, rather than segments, is an example of sandwiching. Large block objects are an appropriate representation object since they are potentially large, but do not have ACLs as segments do. Maintaining the physical attributes of segment objects in the Map, rather than in some higher-level directory object, is another example of sandwiching. The functioning of the Map, unlike the functioning of directory objects, does not depend upon correct behavior of the segment layer.

In this chapter we have shown how the map layer, by treating the attributes of higher-level objects as elements of a property list, can avoid a strong dependence on the type managers for such objects. This design principle is open-ended: the map layer can associate objects and properties, for an arbitrary collection of type managers.

Chapter V

Conclusions and Suggestions for Further Research

5.1 Introduction

The purpose of this thesis has been to present techniques for understanding the structure of computer operating systems. Techniques for achieving modular structure, such as layering and object-orientation, were used in this thesis. The research reported here, however, goes beyond the use of such techniques.

One reason that we strive to achieve a clean, modular structure of a system is to make the system as a whole more amenable to verification. To verify the correct operation of a module, it is necessary to consider intermodule dependencies. This thesis not only suggests how to achieve modular structure, but also presents a point of view that provides for straightforward identification of intermodule dependencies in the context of a case study subsystem. Using this point of view, we can determine which dependencies are necessary and which are superfluous.

The framework presented in this thesis for understanding intermodule dependencies is derived from the LISP world of atomic objects. Atomic objects in the LISP world have bindings and property lists. A binding, in turn, is an atomic object, and each element of a property list is an atomic object. Every atomic object is characterized by its binding and property list. A collection of such objects can serve as a formal model for describing the structure of complex systems. In this object world there is a strong notion of modularity: the behavior of any one object can be characterized completely without any

knowledge of the objects designated by the property list or by the binding. In this thesis, we have considered a virtual memory subsystem to which this notion of modularity is applicable. The VM subsystem has not been implemented; however, it is patterned after virtual memory subsystems of several contemporary, general-purpose operating systems. There are several reasons for the choice of a VM subsystem as the case study. First, it provides a challenge: the actual VM subsystems that serve as a basis for the case study VM are quite complex. Second, the LISP notion of modularity is well-suited to a case study subsystem that includes memory multiplexing facilities and extended type managers. In this thesis, we model memory multiplexing simply as the manipulation of bindings among objects, and we model the mapping from an extended object to a component simply as a list of properties of the extended object name.

5.2 Results

We have shown how the notion of binding and the notion of property list may be used to make intermodule dependencies explicit. Application of the binding notion was considered in chapter III, and application of property lists was considered in chapter IV.

Chapter III presented the point of view that the structure of a system can be simplified if the relations between objects are represented as bindings. If system designers exploit this binding notion, they should be able to identify and eliminate unnecessary dependencies. Since the memory multiplexing model presented in chapter III is a model of objects related by bindings, the multiplexing function should be able to be provided by a collection of modules that exhibit few interdependencies. In chapter III we

pointed out that not one, but rather several, of the layers of the case study VM carry out a multiplexing function. Higher layers multiplex abstract objects supplied by lower layers. Thus the structuring advantages provided by the memory multiplexing model can be applied to several of the VM layers. Additionally, much of the multiplexing mechanism serves as a greatest common mechanism for these VM layers.

Chapter IV focused on the higher VM layers, in which mapping from extended objects to component objects, rather than memory multiplexing, is a common function. We stressed that the subsystem that actually implements such a mapping need not depend upon the subsystems that manage either the extended object or the component objects. It is sufficient for one to regard the component objects (or any attributes that may be just a part of an object) as properties of the object name. Since several of the VM layers described in chapter IV require a mechanism for mapping objects to attributes, a particular subsystem called the map layer can perform the mapping, serving as a greatest common mechanism.

5.3 Comparison of Object Bindings and Property Lists

The binding and property list notions correspond to two somewhat different points of view about interpretation of object names in object-oriented systems. To see this, we consider several scenarios.

In an object-oriented system it is reasonable to expect that databases contain descriptors, or names, of objects. In particular, the name of an object managed by type manager A may exist in a database of type manager B. If B relies on the validity of this name, then it will probably depend on A since A could invalidate the name. If A relies on the validity of the name,

then it will probably depend on B since B manages the database. If both the above conditions exist, then there is a potential for a circular dependency between A and B. There are three other cases, which, by themselves, will not lead to a circular dependency.

First, type manager A may not rely on the validity of the name (and moreover may not know of its existence). In this case A cannot depend on B, assuming there are no other causes for a dependency. This case is typical in computer software systems; for example, the manager of an object depends on the managers of the representation objects.

Second, type manager B may not rely on the validity of the name. In this case, B is maintaining a property list for type manager A. Accordingly type manager B should not depend on type manager A. This case is probably encountered less frequently than the preceding one. A facility for storing and forwarding (but not interpreting) messages is an example of this case. A message is simply a property of a message container name.

Third, neither type manager may rely on the validity of the name. This case corresponds to the situation in which type manager B maintains a binding to an object of type A. Accordingly, there need not be a dependency in either direction. Instances of this case probably occur infrequently in operating systems. The behavior of the object managers of the memory multiplexing model corresponds to this case.

The latter two cases are those in which the maintainer of an object descriptor need not depend on the semantics of the object. This independence characteristic also applies to object managers in the LISP world of atomic objects. Although a situation in which one subsystem maintains a property list that is used by another corresponds only to "one-way" independence, it

should be able to be exploited frequently. A situation in which a subsystem maintains bindings to objects corresponds to a "two-way" independence, but it probably arises only infrequently.

5.4 Remaining Problems and Future Research Directions

The goal of this research has been to suggest a new point of view that may be applied to the structuring of large software systems. In this section we raise some questions about the assumptions underlying the research and about the generality of the results. In addition we suggest directions for future research that may answer some of these questions.

The object-oriented approach taken in this thesis has allowed us to take a strict view of modularity. It has the drawback, however, that it is difficult to apply in some cases. Operations that affect more than one object, for example, cannot be modelled conveniently. In chapter III we defined the fetch and store operations to apply jointly to home and frame objects, since they affect the data bindings of both objects. Consequently the home and frame objects cannot be managed by distinct type managers. Either these two types must be managed by a single, larger type manager or more than one subsystem must serve as a type manager for a given object type. Neither alternative is desirable: in the first case, it may be hard to show that unexpected interactions between the two object types cannot exist; in the second case, a subsystem that depends on one of the contending type managers probably must depend on the other as well. We can conclude that the kind of modularity imposed by an object-oriented view may not always be appropriate for software systems. Further inquiry into the nature of modularity in large software systems is needed.

The structuring methods of this thesis have been applied to a case study virtual memory subsystem. Relevant questions about the applicability of these methods include 1) whether an implemented VM subsystem could actually be structured in this way, and 2) whether these methods have a wider range of applicability -- such as applicability to a file manager node in a distributed system or to the input/output facilities of a computer system. In chapter IV it was argued that a VM design, structured according to these methods, should be able to be carried through to an implementation without sacrificing the structure. The second question, however, has not been considered in this thesis, and is still an open research issue.

Although the notion of correct module operation used in this thesis is informal, it includes some rather strong assumptions. As a result, in several cases one module was declared to be dependent on another even though, in a narrower sense, the former could operate correctly in spite of failures of the latter.

A first approach to specifying correct operation, which we call the "accumulated semantics" approach, states that in order for a subsystem to be correct, not only must it manipulate some set of objects as specified, but all the type managers for objects in that set must (recursively) do so. The accumulated semantics approach focuses on the correct operation of an interface, rather than of any single module.

A second approach to specifying correct operation, which we call the "isolated semantics" approach, states that in order for a subsystem to be correct, it is necessary only that it manipulate some set of objects as specified. Whether the objects in the set behave according to their specification is irrelevant. It is assumed that the subsystem can tolerate

errant behavior of any of the corresponding type managers. Even with this approach, though, there can exist interfaces for which correct operation depends on the correct operation of a collection of subsystems.

At some points in this thesis we have favored the accumulated semantics approach since it is stronger and more generally applicable. Nonetheless, the isolated semantics approach appears to be a useful one. For example, subsystems like the block and frame sublayers, which are independent in this narrower sense, can be implemented and debugged separately. In addition, it should be easier to locate an errant module in a collection of modules that is independent in the isolated sense. Further research into the nature of the correct operation of modular systems can help us better apply a traditionally abstract notion to the engineering of complex software systems.

The goal of this thesis has been to enhance our understanding of modular structure and dependency in computer software systems. While we feel that these research results are applicable to a variety of such systems, further research will be required to determine the scope of these results. We should develop the breadth and depth of our knowledge by improving our models of modularity and dependency. Research on these topics should lead to better methodologies for the design of correct, reliable systems; it should help offset the rising cost of software production; and it should improve our ability to predict the performance of large systems.

BIBLIOGRAPHY

[Bell74] D. E. Bell, L. J. La Padula, "Secure Computer Systems: Mathematical Foundations and Model", The MITRE Corporation, Bedford, Mass., M74-244 (October 1974).

[Benjamin76] A. J. Benjamin, "Improving Information Storage Reliability Using a Data Network", M.I.T. Laboratory for Computer Science Technical Memo TM-78 (October 1976).

[Bensoussan72] A. Bensoussan, C. T. Clingen, R. C. Daley, "The Multics Virtual Memory: Concepts and Design", Communications of the ACM 15, 5 (May 1972), pp. 308-318.

[Bobrow72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", Communications of the ACM 15, 3 (March 1972), pp. 135-143.

[Bratt75] R. G. Bratt, "Minimizing the Naming Facilities Requiring Protection in a Computing Utility", M.I.T. Laboratory for Computer Science Technical Report TR-156 (September 1975).

[Cohen75] E. Cohen, D. Jefferson, "Protection in the Hydra Operating System", Proceedings of the Fifth Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5 (November 1975), pp. 141-160.

[Dennis65] J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", Journal of the ACM 12, 4 (October 1965), pp. 589-602.

[Dennis66] J. B. Dennis, E. G. Van Horn, "Programming Semantics for Multiprogrammed Computations", Communications of the ACM 9, 3 (March 1966), pp. 143-155.

[Dijkstra68] E. W. Dijkstra, "The Structure of the THE Multiprogramming System", Communications of the ACM 11, 5 (May 1968), pp. 341-346.

[England72] D. M. England, "Architectural Features of System 250", Infotech State of the Art Report 14 (Operating Systems), Infotech Information Limited, Maidenhead, Berkshire, England (1972), pp. 395-428.

[Fabry74] R. S. Fabry, "Capability-Based Addressing", Communications of the ACM 17, 7 (July 1974), pp. 403-412.

[Floyd67] R. W. Floyd, "Assigning Meanings to Programs", Proceedings of Symposium in Applied Mathematics, Volume 19, (ed. J. T. Schwartz) American Mathematical Society, Providence, R. I. (1967), pp. 19-32.

[Habermann76] A. N. Habermann, L. Flon, L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", Communications of the ACM 19, 5 (May 1976), pp. 266-272.

[Hoare69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", Communications of the ACM 12, 10 (October 1969), pp. 576-580.

[Huber76] A. R. Huber, "A Multi-Process Design of a Paging System", M.I.T. Laboratory for Computer Science Technical Report TR-171 (December 1976).

[IBM73] International Business Machines Corporation, "System/370 Principles of Operation", IBM Corporation Systems Reference Library GA22-7000-3 (1973).

[Janson76] P. A. Janson, "Using Type Extension to Organize Virtual Memory Mechanisms", M.I.T. Laboratory for Computer Science Technical Report TR-167 (September 1976).

[Jones73] A. K. Jones, "Protection in Programmed Systems", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University (June 1973).

[Jones75] A. K. Jones, W. A. Wulf, "Towards the Design of Secure Systems", Software Practice and Experience 5, 4 (October-December 1975), pp. 321-336.

[Lampson69] B. W. Lampson, "Dynamic Protection Structures", AFIPS Fall Joint Computer Conference Proceedings, Volume 35, AFIPS Press, Montvale, N. J. (1969), pp. 27-38.

[Lampson73] B. W. Lampson, "A Note on the Confinement Problem", Communications of the ACM 16, 10 (October 1973), pp. 613-615.

[Lampson76] B. W. Lampson, H. E. Sturgis, "Reflections on an Operating System Design", Communications of the ACM 19, 5 (May 1976), pp. 251-265.

[Lett68] A. S. Lett, W. L. Konigsford, "TSS/360: A Time-Shared Operating System", AFIPS Fall Joint Computer Conference Proceedings, Volume 33, AFIPS Press, Montvale, N. J. (1968), pp. 15-28.

[Liskov72] B. H. Liskov, "The Design of the Venus Operating System", Communications of the ACM 15, 3 (March 1972), pp. 144-149.

[McCarthy62] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, M. I. Levin, "LISP 1.5 Programmers Manual", second edition, The M.I.T. Press, Cambridge, Mass. (1962).

[Montgomery76] W. A. Montgomery, "A Secure and Flexible Model of Process Initiation for a Computer Utility", M.I.T. Laboratory for Computer Science Technical Report TR-163 (June 1976).

[Multics74] --- "Introduction to Multics", M.I.T. Laboratory for Computer Science Technical Report TR-123 (February 1974).

[Nakazawa72] K. Nakazawa, K. Murata, K. Ishihara, H. Iwakami, H. Horikoshi, H. Nishino, K. Noda, "The Development of the High Speed National Project Computer System", First USA-Japan Computer Conference Proceedings, Hitachi Printing Co., Tokyo, Japan (1972), pp. 173-181.

[Naur66] P. Naur, "Proof of Algorithms by General Snapshots", BIT 6, 4 (1966), pp. 310-316.

[Neumann75] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, A. R. Saxena, "A Provably Secure Operating System", Stanford Research Institute, Menlo Park, Calif. (June 1975).

[Organick72] E. I. Organick, "The Multics System: an Examination of its Structure", The M.I.T. Press, Cambridge, Mass. (1972).

[Parnas72] D. L. Parnas, "A Technique for Software Module Specification with Examples", Communications of the ACM 15, 5 (May 1972), pp. 330-336.

[Parnas76] D. L. Parnas, "Some Hypotheses About the "uses" Hierarchy for Operating Systems", Research Report BS I 76/1, Technische Hochschule Darmstadt, Fachbereich Informatik (March 1976).

[Popek74] G. J. Popek, "A Principle of Kernel Design", AFIPS National Computer Conference Proceedings, Volume 43, AFIPS Press, Montvale, N. J. (1974), pp. 977-978.

[Radin76] G. Radin, P. R. Schneider, "An Architecture for an Extended Machine With Protected Addressing", IBM Poughkeepsie Laboratory Technical Report TR 00.2757 (May 1976).

[Redell74] D. D. Redell, "Naming and Protection in Extensible Operating Systems", M.I.T. Laboratory for Computer Science Technical Report TR-140 (November 1974).

[Reed76] D. P. Reed, "Processor Multiplexing in a Layered Operating System", M.I.T. Laboratory for Computer Science Technical Report TR-164 (June 1976).

[Robinson75] L. Robinson, K. N. Levitt, P. G. Neumann, A. R. Saxena, "On Attaining Reliable Software for a Secure Operating System", Proceedings of the International Conference on Reliable Software, ACM SIGPLAN Notices 10, 6 (June 1975), pp. 267-284.

[Rotenberg74] L. J. Rotenberg, "Making Computers Keep Secrets", M.I.T. Laboratory for Computer Science Technical Report TR-115 (February 1974).

[Saltzer74] J. H. Saltzer, "Protection and the Control of Information Sharing in Multics", Communications of the ACM 17, 7 (July 1974), pp. 388-402.

[Saltzer75] J. H. Saltzer, M. D. Schroeder, "The Protection of Information in Computer Systems", Proceedings of the IEEE 63, 9 (September 1975), pp. 1278-1308.

[Schroeder72] M. D. Schroeder, "Cooperation of Mutually Suspicious Subsystems in a Computer Utility", M.I.T. Laboratory for Computer Science Technical Report TR-104 (September 1972).

[Schroeder75] M. D. Schroeder, "Engineering a Security Kernel for Multics", Proceedings of the Fifth Symposium on Operating Systems Principles, ACM Operating Systems Review 9, 5 (November 1975), pp. 25-32.

[Stern74] J. A. Stern, "Backup and Recovery of On-Line Information in a Computer Utility", M.I.T. Laboratory for Computer Science Technical Report TR-116 (January 1974).

[Sturgis74] H. E. Sturgis, "A Postmortem for a Time Sharing System", Xerox Palo Alto Research Center, Palo Alto, Calif., CSL 74-1 (January 1974).

[Whitmore73] J. C. Whitmore, A. Bensoussan, P. A. Green, D. H. Hunt, A. Kobziar, J. A. Stern, "Design for Multics Security Enhancements", United States Air Force Electronic Systems Division, Technical Report ESD-TR-74-176 (December 1973).

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM 17, 6 (June 1974), pp. 337-345.